Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 86

ABB Corporate Research, Baden-Dättwil*

Systems Group, Department of Computer Science, ETH Zurich†

Smart Eagle

Advanced external monitoring of heterogeneous networks

by
Bram Scheidegger

Supervised by
Yvonne-Anne Pignolet*
and Timothy Roscoe†

July 2013 - February 2013

inf | Informatik
Computer Science

Abstract

The energy sector is undergoing major changes including a trend towards possibly isolated parts of the grid (microgrid) with small power plants producing electrical energy from renewable sources, leading to an unpredictable availability of electricity. To balance the network load, the communication between consumers, producers, sensors and other devices is vital. The backbone of such a self-regulating infrastructure is the data network comprised of various very different network types.

Despite the importance of the data network, there are currently no vendor independent analysis and management tools available tailored for such networks. We present Smart Eagle, a distributed network analysis and monitoring tool capable of dealing with building automation and low-rate wireless personal area networks. We report on our experience in taking measurements for such networks and introduce the architecture of Smart Eagle.

We were able to gather meaningful measurements from both network types without changing the underlying network infrastructure. The evaluation shows that our measurement techniques are comparable or better than tools covering one technology only. Furthermore, we feel that our model and architecture fits the microgrid requirements well, laying the cornerstone to add support for even more network types.

Acknowledgements

I would like to thank my study mentor and supervisor of this Master's thesis, Prof. Timothy Roscoe, for making a project in corporation with ABB corporate Research in Dättwil even possible, and for his great support, time and advice during my study and especially my Master's thesis.

Also, I would like to thank the research team from the C department at ABB Corporate Research Dättwil, in particular my supervisor Yvonne-Anne Pignolet for the original project idea as well as her time and advice during the project. In addition, many thanks to Ettore Ferranti and Thanikesavan Sivanthi for the insightful discussions and their suggestions to improve my work. I am also in debt to Christian Winnewisser from ABB Stotz-Kontakt in Germany who answered a lot of KNX related questions.

Finally, I would like to thank my girlfriend and parents for their support and patience during my whole study time.

Dättwil, February 2013

Contents

1 Introduction **1**
- 1.1 Smart grids . . . 1
- 1.2 Motivation . . . 2
- 1.3 Aim . . . 4
- 1.4 Document structure . . . 4
- 1.5 Terminology . . . 4
- 1.6 Abbreviations . . . 5

2 Background **7**
- 2.1 KNX . . . 7
  - 2.1.1 Introduction . . . 7
  - 2.1.2 Topology . . . 7
  - 2.1.3 Addressing scheme . . . 9
  - 2.1.4 Layer model . . . 10
  - 2.1.5 TP 1 Acknowledgement behavior . . . 11
  - 2.1.6 KNXnet/IP . . . 12
- 2.2 IEEE 802.15.4 . . . 13
  - 2.2.1 Introduction . . . 14
  - 2.2.2 Addressing . . . 14
  - 2.2.3 Data frame format . . . 15
  - 2.2.4 Security and encryption . . . 15
  - 2.2.5 Link layer acknowledgements . . . 16
- 2.3 Ethernet . . . 16
  - 2.3.1 Ethernet and IP . . . 16
- 2.4 Related work . . . 16
  - 2.4.1 Ethernet and IP network measurements . . . 17
  - 2.4.2 Smart grid network measurements . . . 18
  - 2.4.3 KNX network measurements . . . 18
  - 2.4.4 IEEE 802.15.4 and Zigbee network measurement . . . 19
  - 2.4.5 Distributed smart grid control . . . 19
- 2.5 Related software . . . 20
  - 2.5.1 KNX . . . 20
  - 2.5.2 IEEE 802.15.4 . . . 22

3 Project overview **23**
- 3.1 ABB smart grid demo lab . . . 23
  - 3.1.1 Network structure . . . 23
- 3.2 Architecture . . . 24
  - 3.2.1 Motivation . . . 24
  - 3.2.2 Structure . . . 25
- 3.3 Application functionality . . . 27
  - 3.3.1 Measurement functionality . . . 27
  - 3.3.2 Graph view . . . 27
- 3.4 Shared modules . . . 27

3.4.1 Logging facility . . . 28
3.4.2 Locator beacon . . . 28
3.5 Control unit architecture . . . 30
3.5.1 Communication . . . 30
3.5.2 Transactions . . . 31

4 IEEE 802.15.4 probe 35
4.1 Architecture . . . 35
4.2 Network adapter . . . 36
4.2.1 Econotag hardware platform . . . 36
4.2.2 Contiki . . . 38
4.2.3 Sniffing . . . 41
4.2.4 Round trip time measurement . . . 41
4.2.5 Acknowledgements . . . 43
4.2.6 Active device discovery . . . 45
4.2.7 Link quality . . . 49
4.3 Interconnect . . . 49
4.3.1 Contiki interconnect I/O . . . 49
4.3.2 Java interconnect I/O . . . 50
4.4 Control unit . . . 50
4.4.1 Architecture overview . . . 50
4.4.2 Measurement transactions . . . 51
4.4.3 IO handler service . . . 51
4.4.4 Automatic deployment . . . 52

5 KNX probe 53
5.1 Architecture . . . 53
5.2 Network adapter . . . 53
5.3 Interconnect . . . 53
5.4 Control unit . . . 53
5.4.1 Architecture overview . . . 54
5.4.2 Gateway discovery . . . 54
5.4.3 Sniffing . . . 55
5.4.4 Measurement functionality . . . 56
5.4.5 Concurrency . . . 57
5.4.6 Simulator . . . 57

6 Measurement application 58
6.1 Architecture overview . . . 58
6.2 Data storage . . . 58
6.2.1 Root node . . . 59
6.2.2 Child node . . . 60
6.3 Measurement functionality . . . 60
6.3.1 CuManager . . . 60
6.3.2 Measurement . . . 61
6.3.3 Filter & inference . . . 62

6.4 Graphical user interface . . . 62
6.4.1 Implementation . . . 63
6.4.2 Graph view . . . 65

7 Evaluation **66**
7.1 Application & system . . . 66
7.1.1 Stability . . . 66
7.1.2 Transactions . . . 67
7.1.3 GUI . . . 67
7.2 IEEE 802.15.4 . . . 67
7.2.1 Device discovery . . . 68
7.2.2 Monitoring . . . 69
7.2.3 Channel scan . . . 69
7.3 KNX . . . 70
7.3.1 Topology discovery . . . 70
7.3.2 Discovery accuracy . . . 70
7.3.3 Performance & threading . . . 71
7.3.4 Connectionless discovery . . . 71
7.3.5 Data source combination . . . 72
7.3.6 Link layer analysis . . . 72
7.4 Known issues . . . 74

8 Conclusions **75**
8.1 Future work . . . 75
8.1.1 Improvements . . . 75
8.1.2 Extensions . . . 76
8.1.3 Integration . . . 79

9 Appendix **84**
9.1 Setup and deployment . . . 84
9.1.1 Default gateway . . . 84
9.1.2 Eclipse setup . . . 84
9.1.3 IEEE 802.15.4 probe . . . 84
9.1.4 KNX probe . . . 85
9.1.5 Measurement application . . . 85
9.2 IEEE 802.15.4 interconnect protocol . . . 85
9.3 Intermediate network protocols . . . 86

1 Introduction

Computer networks are an integral part of today's society and our dependency on them is growing continuously. Examples of such networks can be found in the telecommunication sector or the financial industry. Currently, the trend is going towards an even tighter integration of systems, for instance in building automation: connecting various sensors and actuators to a network allows components like wind, light and temperature sensors to interact with the blinds and ventilation system.

The importance of data networks require us to successively monitor them to ensure proper operation and detect failures, performance problems or malfunctions. Furthermore, when setting up new networks, we need to know whether they fulfill their specifications which may for example depend upon the level of interference in case of wireless networks. Without network measurement and monitoring tools at hand we would be blind.

In this master's thesis we focus on measurements and monitoring of *heterogeneous* networks in an uncooperative environment. We define a *heterogeneous* network as a network consisting of a multitude of different network technologies and we do not require that communication takes place using the popular Internet Protocol (IP) only. By *uncooperative* we denote an environment where no additional software can be installed.

We emphasize on communication networks for smart grids as an example of such heterogeneous networks and tailor our analysis and tools towards smart grid environments. However, the approach taken is applicable to other heterogeneous networks as well.

1.1 Smart grids

The invention of electricity is a milestone in the development of mankind that we take for granted nowadays. Today's national grids are mostly composed of large power plants (hydro-electric power stations, nuclear power plants, etc.) and a transmission and distribution network which delivers electricity to consumers.

At the moment the energy sector is undergoing major changes due to the need to renew the old grid infrastructure and due to the increasing popularity of renewable energy sources like wind and solar energy. These types of power generators are typically deployed at smaller scale which leads to a distributed instead of a centralized generation of electrical energy. To avoid destabilizing the grid, the power output of small power plants like wind-farms need to be controlled to ensure proper grid operation [1].

Compared to large power plants, the energy production of wind and solar power generators is weather dependent and hence unable to provide electricity on demand. One way to tackle this problem is by adding additional components for energy storage which can be achieved by dedicated units like batteries or by reusing for example the batteries of an electrical car. Hence there is no strict separation of producer and consumer within the grid anymore (e.g. a battery charging or discharging) which leads to new challenges in grid management

and grid stabilization. To find possible solutions to deal with this increasing complexity, smart grids are currently a hot topic in research.

One key idea of a smart grid is to provide two-way, digital communication to devices which enables the interaction between sensors and actuators distributed throughout the grid and even within households [2]. Apart from this and the integration of renewable energy sources, Zhenhua Jiang at al. present a set of goals associated with a smart grid deployment [3]:

- Flexibility: embrace future extensions of the grid and cope with new types of energy markets.
- Intelligence: the grid is not only controlled by a central command and control station but includes some local control functions itself.
- Resiliency: the intelligence of the smart grid is used to implement self-healing capabilities to recover from blackouts.
- Customization: provide the consumer with several options (e.g. pricing schema's) to adapt the grid to its needs.

To illustrate the benefits of smart grids we briefly examine two typical use cases in more detail.

Balance power consumption As mentioned earlier, the availability of electricity becomes dependent on weather conditions when using wind and solar energy. The availability or price information is communicated to the customer (making use of the smart grid two-way communication capabilities) and the end-user devices can react on this, for example by not heating up the boiler if electrical energy supply is short [4].

Microgrid islanding A local power failure can quickly become a huge problem (for example due to cascading overloads) and in the end affect millions of people [5]. To prevent such a cascade, a smart grid shall perform real time monitoring and detect possible problems in advance.

However, this approach may not always succeed. To prevent the grid from failing entirely in such an event, it should breakup into self-sustaining *islands*. The power outage affects far less people which is especially useful for critical infrastructure. An example deployment of a microgrid is located in the Santa Rita Jail in California [6].

1.2 Motivation

Our primary focus are *microgrids*, defined as a group of electricity producers, storage and consumers attached to the main power grid having the capability to switch into microgrid islanding mode. Typical scenarios for microgrid deployments are hospitals, campus, settlements etc. A difference between the main grid and microgrids is the granularity of control: a microgrid has fine grained

control over various components within a building (e.g. heating system, lights, etc.) whereas large scale networks operate on entire areas, e.g. streets.

Within such a microgrid, the communication infrastructure is essential to enable the interaction between sensors, actuators, producers and customers. During daily operation, the network is mainly used to control the heating system, lights, power levels, etc. In case of a power failure, the data network transports essential information to stabilize the microgrid and keep important systems operational. As the data network is the backbone of all these operations, it is vital to keep it operational. This requires a system which assists in deployment, operation and maintenance of the underlying communication infrastructure.

In the next paragraphs, we identify the target audience of such a network measurement product and why it is valuable to a customer. Afterwards we argue about the limitations of software deployment to support our approach.

Target audience Microgrid operators have to monitor their own infrastructure as they have to detect issues and react on them appropriately. If they outsourced part of their data network (e.g. to a telecommunications provider), they may want to oversee that the service level agreement is satisfied.

Measurement and monitoring is useful for manufacturers of smart grid communication components as well to asses if their solutions meet the specifications. Such tests are not limited to laboratory experiments but involve field studies as well.

Business value The communication infrastructure is vital to the operation of a smart grid (Subsection 1.1). Monitoring the underlying communication network is worthwhile the effort as power outages result in major economic damage [5]. For microgrids, a typical example is a company facility and its associated data center: not being able to prioritize the data center could result in a long down-time or even data loss.

Diversity From a network point of view, a smart grid is a large, heterogeneous network consisting of a multitude of different devices. The network size is mainly determined by the number of attached devices; for microgrids we estimate a few thousands. The devices participating in communication can be grouped into two main categories:

- Electricity supplier devices: power generation, power storage units, sensors and actuators to control the grid.
- Building automation: end-user devices allowing fine-grained control. These are dedicated units, for example smart meters or integrated modules (e.g into heating systems).

As devices are delivered by various manufacturers, it is likely that custom software deployment is limited or unavailable. Therefore, we choose a least invasive approach and perform our measurements by adding devices only.

1.3 Aim

Our goal is to provide a software demonstrator named *Smart Eagle* with a user interface combining all relevant measurements. As the networks under analysis are very different from each other, a software architecture capable of dealing with this diversity is presented. Smart Eagle implements the components necessary to analyze a home automation bus called KNX and wireless networks built on top of the physical and network layer of IEEE 802.15.4. These are typical networks for building automation and control. As the knowledge about the required techniques to conduct measurements on KNX and IEEE 802.15.4 is limited, we devise and implement new network analysis algorithms.

1.4 Document structure

Chapter 2 provides background information about the networks we analyze and gives an overview about related work in the area of network and smart grid measurements. A project overview is presented in Chapter 3, describing our test network (ABB smart grid demo lab), the architecture and features of Smart Eagle, a set of common modules used throughout the system and the basic control unit architecture.

Chapters 4 and 5 give an in-depth description about the design and implementation of the IEEE 802.15.4 and KNX measurement modules. In particular, the various measurement mechanisms and pitfalls are described. The measurement application including the graphical user interface is presented in Chapter 6.

In Chapter 7, we evaluate our software in terms of design and measurement precision. Chapter 8 concludes our work and provides an outlook to future research.

1.5 Terminology

Active - passive By *passive* network measurement, we obtain our measurements through observation only. When using *active* measurement techniques, we interact with the network, e.g. by sending probing traffic.

External Network measurement functionality can either be implemented on existing devices by extending their functionality or added separately which means that there is dedicated hardware for measurement purposes only.

The separately added measurement hardware can be an integral part of the network, i.e. it has the same network access capabilities as all other nodes. An example for such an access capability is a pre-shared encryption key. Another option is that the measurement node merely possesses the required network interface and a network stack according to the specifications.

The approach of adding a separate piece of hardware (dedicated for measurement) which only implements the required network interface and a network stack according to the specifications is what we call *external*.

Node By a *node*, we refer to the physical device attached to that particular network (not the network port). This has to be clearly distinguished from an address because one physical device can have multiple network addresses.

The word *node* is also used when describing data structures. However, this becomes clear from the context.

Network element We define a network element as a device required for network operation. Routers and switches are typical examples in an IP network.

Latency *Latency* is defined as the time for one message to traverse the network from source to destination. This is also known as *delay*. The latency is not always half the RTT because the network link may be asymmetric. Latency can be measured on different layers, e.g. link layer, network layer or application layer.

Round trip time The *round trip ti*me *(RTT)* is defined as the time for one message to traverse the network from source to destination plus the time the response takes to travel from the destination back to its source. Like latency, the RTT can be measured on different layers as well.

Uncooperative In an *uncooperative* environment, the network nodes under test do not possess additional software to participate in the measurement process.

1.6 Abbreviations

Summary of the most widely used acronyms.

802.15.4	The terms IEEE 802.15.4 and for short 802.15.4 are used interchangeably. More details are provided in Subsection 2.2.
ACK	Acknowledgement
API	Application programming interface. The interface a software component presents to the outside world.
cEMI	Common External Message Interface. A medium independent KNX frame.
CSMA	Carrier sense multiple access. A medium access protocol that ensures the absence of traffic before the transmission starts.
DHCP	Dynamic Host Configuration Protocol. Allows clients in an IP network to automatically obtain their IP address from a central server.
ETS	Engineering Tool Software. Official, vendor-independent project engineering tool for KNX, developed by the KNX consortium.

IP	Internet Protocol. A widely used network layer protocol.
JAXB	Java Architecture for XML Binding. Offers marshalling and unmarshalling of Java objects using XML as intermediate format.
KNX	Network communications protocol for home and building automation systems (Subsection 2.1).
MVC	Model–view–controller pattern. A widely used software architecture pattern to separate the GUI code from the program logic and data.
NAK	A negative acknowledgement (not acknowledged).
NUT	Network under Test. The network we are analyzing.
RDC	Radio Duty Cycling. Defines when a radio chip is permitted to sleep for the purpose of saving energy.
RTT	Round Trip Time. The time it takes for a signal to travel from its source to its target and backward (Subsection 1.5).
TE	Transaction executor. The Smart Eagle framework for transaction based measurements.
UART	Universal Asynchronous Receiver/Transmitter. This interface is used for serial communication on a serial port.
XML	Extensible Markup Language. A widely used language for encoding documents or data structures in a common format.
WTE	WebTransactionExecutor. A part of the Smart Eagle transaction framework running on the control unit to execute a measurement web request.

2 Background

In this chapter, we provide the required background knowledge for this thesis. We start by introducing the two types of network we are analyzing. Next, we briefly look at Ethernet networks and in the end we discuss related work and related software.

2.1 KNX

In this subsection, we present the background knowledge for KNX. We start by introducing the topology and addressing schema. Afterwards, we discuss the network layers of KNX and in particular the link layer ACK behavior for the KNX twisted pair medium. In the end, we briefly review the KNXnet/IP protocol which we use to access the KNX bus.

2.1.1 Introduction

KNX is a home automation system designed to interconnect a wide range of building automation components like heating control, lights, blinds, etc. and was ISO standardized in 2006 [7, 8]. The KNX specification defines the network from the physical media (twisted pair, power line, wireless) up to the application layer. KNX does not only specify how to transport data but also defines a set of commands for each type of device as well. Hence, all KNX certified light switches use the same set of commands which makes the KNX deployment manufacturer independent.

Other home automation systems Along with KNX, there are two other major building automation systems on the market: the Building Automation and Control Networking Protocol (BACnet) and the Local Operation Network (LonWorks) [9]. They are used throughout many countries worldwide whereas KNX is mainly used on the European market. The principle idea of all three building automation systems is the same: supporting a variety of different communication media and specifying the interaction of devices.

BACnet development started 1987 and it standardizes a small number of network types, one of them is Point-to-Point to support dial-up communication. LonWorks is comprised of a communication protocol (LonTalk, standardized 1999), a dedicated controller and a network management tool.

2.1.2 Topology

The KNX topology consists of a three level hierarchy where each level can contain network elements and devices. Network elements are devices which are required for network operation, like routers or bridges.

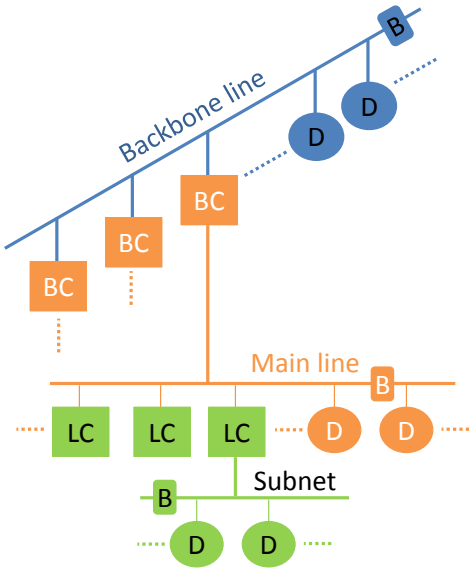

Figure 1: Topology of a KNX network: a three level hierarchy where each level except the subnet can hold both devices and network elements. Abbreviations: Backbone Coupler (BC), Line Coupler (LC), Device (D) and Repeater or Bridge (B).

Device (TP 1)	Functionality	Individual address	Selective ACK	Hop count
Repeater	Interconnect segments within the same hierarchical level	No, except for configuration	No	dec
Bridge	Interconnect segments within the same hierarchical level	No, except for configuration	No	keep
Router	Interconnect different hierarchical levels (backbone or line coupler)	Yes	Yes	dec

Table 1: Summary of KNX network element capabilities. Routers are used to interconnect different hierarchical levels whereas repeaters and bridges are placed within the same hierarchical level. The ACK behavior is reviewed in detail in Subsection 2.1.5. For our purpose, we are not interested whether the hop count is decreased (dec) nor not.

Network elements KNX offers three types of network elements with different capabilities (Table 1). KNX routers are used to establish the topology. They are equipped with filtering capabilities for selective traffic forwarding.

Because twisted pair lines in KNX can get long in terms of cable lengths (up to 3000 m), KNX divides a line into at most four segments. In between these segments, a *TP1 repeater* or *TP1 bridge* is installed to forward network traffic and providing electrical isolation (TP1 also acts as a power supply for attached devices).

Hierarchy Each KNX network has one *backbone line* (first level in hierarchy) and at most 15 *main lines* attached to it (Figure 1). The connection between the backbone line and the main line is established by *backbone couplers*. In addition, a the backbone line can hold up to 255 devices.

A main line (second level in hierarchy) can hold 255 devices plus 15 *line couplers* introducing the third level in the KNX hierarchy called a *line* or *subnet*. A subnet can hold only up to 255 end devices, but no more network elements.

2.1.3 Addressing scheme

A KNX address is 16 bit long and split into three groups usually represented as a decimal number ([10], Section 3/3/2). There are two basic types of addresses: *individual addresses* and *group addresses*.

Group addresses Group addresses are represented by a slash separator (e.g. 1/2/3) and configured during setup. The address is split into two groups of 8 bits each or in a group of 8 bits, 3 bits and 5 bits (configuration dependent). To explain group addressing, we consider the following example: an open-plan office with a set of lights and a light switch including dimming functionality.

Each group address corresponds to a certain action (for example dimming the lights). One group address can be shared by multiple devices (all the lights) and one device can have multiple group addresses (lights: turn on, turn off, dim).

Most of the communication on the KNX bus is through group addresses which are multicast messages. Using group communication is sensible in the application area of KNX (building automation). Consider someone pressing the light switch: instead of sending a separate message to all attached lights, one group message is sent letting the lights turn on at the same time and saving bandwidth. It is therefore not surprising that KNX is intrinsically designed for multicast transmission.

Individual addresses Each device attached to the bus (i.e. the KNX network) has a unique address within the KNX network called an individual address. Compared to group addresses, individual addresses are separated by dots (e.g. 1.2.3). There is a frame header flag to distinguish between individual and group addresses. The individual address corresponds to the device position in the KNX topology simplifying routing and filtering. The three parts of the address are

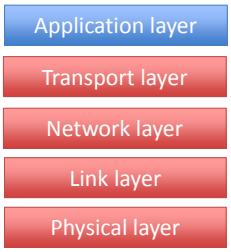

Figure 2: KNX layer model: based on OSI model but session and presentation layer are empty. The layers marked red are partially medium dependent.

(from left to right): area identifier (4 bits), line identifier (4 bits) and device identifier (8 bits).

Network elements are assigned reserved addresses: backbone couplers end with "0.0" (i.e. 1.0.0 to 15.0.0) whereas line coupler addresses end with one zero (for example 1.1.0 to 1.15.0).

2.1.4 Layer model

The layering model of KNX is compliant with the OSI model [11]. Because KNX is defined for different media types, the physical, data link, network and transport layer are partially medium dependent (Figure 2).

Physical layer The specifications of the KNX association define three different physical media types ([10], Section 3/1):

- Twisted pair (TP 1): most widely deployed KNX medium. It can act as power supply for attached devices and offers half duplex communication with a data rate of 9600 bit/s. TP 1 uses the CSMA/CA protocol for medium access.
- Power line (PL 110): data transmission across the 230 V or 400 V power network commonly used in buildings and industry. PL 110 has a data rate of 1200 Kbit/s and uses a CSMA protocol for medium access.
- Radio Frequency (RF): short range wireless communication in the 868,3 MHz band with a bandwidth of 600 kHz. The transmission rate is implementation dependent.

Data link layer The data link layer implements point-to-point transmission within a subnetwork ([10], Section 3/3/2). It includes a medium access protocol and a retransmission algorithm. KNX supports data link layer ACKs to confirm successful frame transmission (Subsection 2.1.5). Upon frame arrival, the data link layer checks whether the frame is corrupted, unpacks the uncorrupted frame and passes it to the next layer.

Because KNX supports different media types, the standard specifies a medium independent part of the data link layer and a medium dependent part. The medium dependent part defines for example medium access and frame formats. The data link layer services and frame priority levels are specified in the medium independent part.

IP integration (KNXnet/IP) is implemented on data link layer: KNX frames are transmitted as payload of an IP packet (Subsection 2.1.6). The IP integration is used for external access or to interconnect bus systems.

Network layer It extends the data link layer by offering communication across different subnetworks ([10], Section 3/3/3). This is achieved through routers (Subsection 2.1.2).

Transport layer On top of the network layer, the transport layer defines five different communication modes ([10], Section 3/3/4). For our purpose, we summarize them into two categories: *Connectionless* and *Connection-oriented*. As the name already suggest, connectionless communication allows to interact with a KNX device without first establishing a connection whereas in connection-oriented mode we first establish a connection, then perform the operation and disconnect in the end. The implementation of connectionless communication is not mandatory by the KNX specifications [12].

Application layer The session and presentation layer are not used in KNX ([10], Section 3/3/5 and Section 3/3/6). The application layer describes a set of services for different connection modes ([10], Section 3/3/7). In a nutshell, the application layer offers functionality to access device memory, serial number, device descriptor, etc. It is the only fully medium independent layer of KNX.

Management Procedures Management procedures are defined on top of the application layer and of special interest for our purpose. They specify device independent functionality applicable to the entire KNX bus such as scanning for in-use network addresses ([10], Section 3/5/2).

2.1.5 TP 1 Acknowledgement behavior

In this subsection, we discuss the link layer ACK behavior of TP1 ([10], Section 3/2/2). A device receiving a telegram can respond in four different ways (Table 2). We explain the KNX ACK behavior by starting with a simple line and then add network elements successively.

Simple line To better understand the semantic properties, we first look at a simple case: a line (e.g. backbone line) containing only devices (no network elements). When getting an ACK, two cases have to be distinguished:

Response	Description
ACK	Telegram destination address matches group or device address and telegram received successfully.
NAK	Telegram destination address matches group or device address but telegram is corrupted on link layer.
BUSY	Telegram destination address matches group or device address but telegram cannot be received because the buffer is full.
IGNORE	Telegram destination address does not match.

Table 2: The four possible link layer responses of a KNX device ([10], Section 3/2/2). For multicast and broadcast, the signals on TP 1 may overlap and the specifications define the outcome.

- Unicast message: ACK if the target device understood the message, NAK otherwise. The target device is the only one eligible to respond to this message.
- Multicast and broadcast: we do not get a distinguishable ACK from each device because the ACK signal is superimposed. The KNX system specifications define that an NAK signal overrides any number of ACK signals.

Router Adding a router to a line does not change the semantics because frames are only ACK if the router is in charge of forwarding the frame according to its topological position (selective ACK). For a group addressed frame, the router consults the filter table which is generated and uploaded to the router during deployment. However, the router sends the ACK on behalf of the device on the other side. Hence, the ACK is not really originated from the target device itself.

Bridge / repeater In contrast to a router, a repeater or bridge within a line completely changes the semantics because it does not support selective ACK. In other words: a repeater (or bridge) ACKs every frame on behalf of a device which may or may not exist.

2.1.6 KNXnet/IP

KNXnet/IP is an extension to the KNX protocol specifying how to encapsulate a KNX frame into a UDP packet ([10], Section 3/2/6). The KNX telegram is in cEMI format which is a KNX medium independent frame format ([10], Section 3/6/3). The KNXnet/IP protocol contains the cEMI frame as payload and adds the following headers ([10], Section 3/8/2):

- Header length: fixed but added to allow future protocol extensions.
- Protocol version: version of the KNXnet/IP protocol.

- KNXnet/IP service: different functionalities of KNXnet/IP are implemented as services, for example: searching for KNXnet/IP gateways or obtaining the capabilities of a KNXnet/IP gateway.
- Total length: total length of a KNXnet/IP frame in octets (including all headers and the cEMI frame).

The KNXnet/IP gateway is the interface between the IP network and the KNX bus and listens on port 3671 for incoming UDP traffic.

To understand the protocol between an IP client (e.g. a PC connected to the KNXnet/IP gateway) and the KNX bus, we consider the following example: the IP client sends a data frame to the bus and the bus responds with a link layer ACK. The KNX frame is transmitted between IP client and bus as a *tunneling request* (the tunneling request already contains the data) and the destination confirms the tunneling request with a *tunneling ACK* (which is *not* the link layer ACK from KNX). In our example, the following steps are executed:

1. IP client $\longrightarrow$ Gateway: tunneling request containing a cEMI frame
2. Gateway $\longrightarrow$ IP client: tunneling ACK
3. Gateway $\longrightarrow$ Bus: forward frame on KNX network
4. Bus $\longrightarrow$ Gateway: Receive link layer ACK from bus
5. Gateway $\longrightarrow$ IP client: tunneling request containing a cEMI ACK frame
6. IP client$\longrightarrow$ Gateway: tunneling ACK

Alternative ways to access the KNX bus from outside (e.g. from a PC) are KNX USB or KNX serial interfaces.

Monitoring modes KNXnet/IP gateways support two monitoring modes: *bus monitoring* and *group monitoring*. In bus monitoring mode, all link layer traffic on the bus is sniffed (including ACK) whereas in group monitoring mode only group communication is forwarded to the IP client. However, not all KNXnet/IP gateways support the bus monitor mode.

The sniffing capabilities also depend on the position of the KNXnet/IP gateway in the bus and the filtering configuration of the routers. If the KNXnet/IP gateway is for example located in the backbone line, we can only sniff the traffic which is exchanged on the backbone line, i.e. between devices on the backbone line or between backbone couplers.

2.2 IEEE 802.15.4

In this subsection, we present background information about IEEE 802.15.4 networks. After a short introduction, we discuss the two different addressing modes and the data frame format. Next, we briefly review the security features and ACK behavior of IEEE 802.15.4.

2.2.1 Introduction

IEEE 802.15.4 is a low-rate wireless network technology enabling low-power and low-cost communication [13]. The specification is publicly available and defines the physical layer and the link layer. The link layer defines the frame format and the medium access control scheme. Medium access control is either slotted or unslotted CSMA/CA or TDMA, this depends on whether the network is operated in beacon or non-beacon enabled mode.

ZigBee and 6LoWPAN are popular personal area network technologies defined on top of IEEE 802.15.4 (Figure 3). In Zigbee, the network layer covers for instance routing or security aspects. The application layer defines an application framework allowing the usage of a standardized set of commands for a specific group of devices (similar to KNX) [14].

The main purpose of 6LoWPAN is to act as a bridge between IPv6 and the 802.15.4 link layer enabling convenient use of all protocols on top of IP. This bridging functionality is necessary because IEEE 802.15.4 networks have a low data rate and a short frame length (127 bytes) compared to IPv6. To allow IP traffic anyway, 6LoWPAN offers IP header compression and packet fragmentation. There are various free and commercial implementations of 6LoWPAN available [15].

2.2.2 Addressing

802.15.4 supports two different addressing modes: *long* addresses (also called *extended* addresses) and *short* addresses. The extended addresses are unique (worldwide), 64-bit long and assigned by the device manufacturer. If a personal area network (PAN) coordinator exits, it may assign a short address to the device after it joined the network ([16], Subsection 7.3.1 and 7.3.2).

Short addresses are 16-bit long and considered local addresses (valid within their network). The addressing mode of each frame is defined within the frame control header. Broadcasts are sent using 0xffff in short addressing mode ([16], Subsection 7.2.1.4).

Different networks are distinguished by their 16-bit long PAN identifier. To further separate different networks in range of each other, 802.15.4 supports

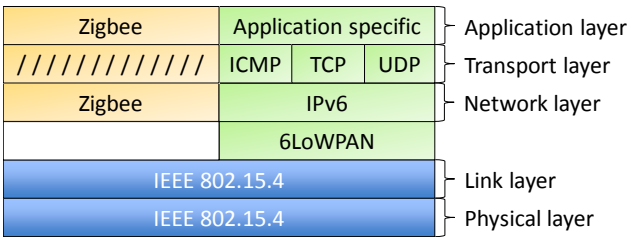

Figure 3: IEEE 802.15.4 serving as low level protocol to implement the higher layer network stacks on top of it, for example ZigBee and 6LoWPAN.

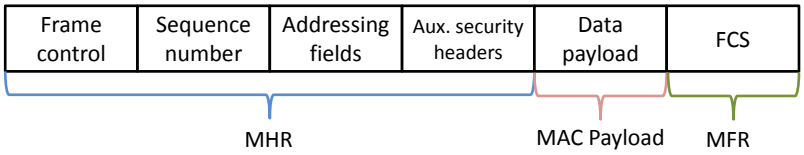

Figure 4: IEEE 802.15.4 data frame format. We are mainly interested in the frame control field, the sequence number and the addressing information. Abbreviations: MAC header (MHR), MAC footer (MFR).

up to 16 different radio channels. Communication between networks having a different PAN identifier is possible whereas devices using a different radio channel are completely isolated. Setting the PAN identifier to 0xffff results in a broadcast to all PANs in range operating on the same channel ([16], Subsection 7.2.1.3).

2.2.3 Data frame format

An IEEE 802.15.4 data frame is split into three major parts (Figure 4): MAC header (MHR), MAC payload and the MAC footer (MFR).

Beside of the addressing information, the MHR carries a sequence number and frame control block. For our purpose, the most important frame control flags are:

- Frame type: identifies the type of this frame, for example data frame or acknowledgement frame.
- Security enabled: enable MAC layer security features to encrypt the payload ([16], Subsection 7.2.1.8). If security is enabled, an auxiliary security header is present.
- Acknowledgement request: set to request a link layer ACK.
- Addressing mode: the source and destination address are configured separately. They can be either short or long addresses.

The payload field is of variable length. The maximum frame size is limited to 127 bytes ([16], Subsection 6.4.1). The MFR contains the frame check sequence (FCS) which is a 16-bit CRC checksum.

2.2.4 Security and encryption

As previously mentioned, the 802.15.4 security features protect the frame payload against eavesdroppers by encrypting it. The MAC header is only protected against tampering by an integrity code and hence always provides us with valuable information like source address, destination address and PAN identifier ([14], Subsection 4.2.2).

Encryption can be implemented on higher layers as well, for example by Zigbee on the network layer. This allows an eavesdropper to read the Zigbee header but the payload is still protected.

2.2.5 Link layer acknowledgements

Setting the acknowledgement request flag in the frame control block automatically causes the receiver to send a link layer acknowledgement if certain conditions are met ([16], Subsection 7.5.6.2). In particular, the destination PAN identifier and destination address must match. Broadcasts are not acknowledged.

Acknowledgement frames only contain a frame control block, the sequence number and a checksum. As the acknowledgement does not contain a source address, it is mapped to the corresponding data frame by its sequence number only.

2.3 Ethernet

As a third network type, we briefly introduce Ethernet. We only use this network to transport our measurement information.

2.3.1 Ethernet and IP

Ethernet (IEEE 802.3) is one of the most widely used networking standards today, specifying the physical and link layer for media types like twisted-pair cabling or power-line. One popular example is 802.3ab [17] which specifies 1 Gbit/s Ethernet over twisted-pair cabling which is used in homes, companies and even data centers.

Ethernet, more specifically the link layer specification of IEEE 802.3, defines only the communication between two directly connected devices. On top of Ethernet, the Internet Protocol (IP) is commonly used to handle communication across several subnetworks. However, in this thesis, we focus our analysis on KNX and 802.15.4 because there is already extensive research on IP networks (Subsection 2.4.1).

2.4 Related work

To the best of our knowledge, there is currently no research towards an integrated solution for measurements in heterogeneous networks. However, the idea of distributed network measurement is already well established. One approach is to deploy nodes in addition to the network to gather data (external nodes) and the other by embedding additional measurement functionality into the nodes. The following subsections contain various examples of both approaches used to acquire different kinds of network information.

2.4.1 Ethernet and IP network measurements

Network measurement in Ethernet and IP based networks in general has already been studied intensively by the research community and resulted in a wide variety of freely available tools. This supports us in our decision to focus our analysis on KNX and Zigbee although IP networks play a major role in a smart grid data network as well.

Bandwidth Prasad et al. cover a wide range of metrics, techniques and tools for bandwidth estimation in Ethernet and IP based networks [18]. The survey paper clearly defines measurement terms, for example differentiating various bandwidth-related metrics. 16 different publicly available tools for bandwidth estimation together with the corresponding measurement metric and methodology are described.

RTT and loss-rate The *ping* utility available on most Microsoft Windows and Linux based installations is a valuable network diagnostic tool [19]. Its functionality is based on the echo request and echo reply functionality of the Internet control message protocol (ICMP) [20]. It is often used by network administrators to determine connectivity, RTT and packet loss rate.

As ping cannot measure the one-way loss rate, Stefan Savage developed a utility named "sting" [21]. It makes use of TCP features like fast retransmit to deduce in which direction the packet loss happened without requiring support from the remote host. By applying sting to popular and random web servers, a significant asymmetry between the forward and backward loss rate has been discovered.

QoS Strohmeier et al. present a distributed QoS measurement approach to assess the performance of AQUILA [22]. AQUILA is QoS architecture running on top of IP to guarantee end-to-end QoS parameters for end-user applications. The measurements are either performed actively by sending a probing flow or passively by gathering network data. As an improvement, they suggest to associate GPS coordinates to the measurement probes to enable localization. Using GPS data could be of interest for our distributed measurement approach as well to locate our measurement nodes distributed throughout several buildings.

Topology discovery Microsoft has implemented an Ethernet link layer network topology discovery feature based on the cooperation of machines running Microsoft Windows Vista or higher [23]. The methodology was proposed by Richard Black et al. from Microsoft Research Cambridge and they do it without using the simple network management protocol (SNMP) protocol. SNMP is used to query and configure network devices. Various other methods for Ethernet topology discovery not limited to Windows machines are presented in a survey by Ahmat [24].

IP networks are often composed of multiple subnetworks interconnected by routers. Many different methods have been proposed to detect the network

layer topology, ranging from querying routers to sending probing packets [25]. A famous example tool to determine the path from a source node to its destination is traceroute which makes clever use of the Time-to-Live (TTL) field in the IPv4 header. As KNX uses a hop count value as well (Table 1), this could be an interesting idea to discover network elements having no individual address.

2.4.2 Smart grid network measurements

Companies selling smart grid product suites often integrate some measurement functionality into their products as they are in full control over the software deployment on the devices. However, the companies only provide vague information about the details concerning network measurements.

Silver Spring Networks is a supplier of networking equipment for power grids. The UtilityIQ Network Element Manager is part of their product portfolio providing remote monitoring and diagnostic functionality [26]. All smart meters compatible with UtilityIQ are required to have a Silverspring communication module on board to enable network monitoring.

IBM offers a framework and management solution named "IBM Intelligent Metering Network Management" for network discovery, topology visualization, root cause analysis and remote device configuration [27].

Apart from these companies, we looked at Nokia-Siemens, Tropos and Infosys but did not find specific information about network measurement.

2.4.3 KNX network measurements

As KNX is an industrial network type and requires special hardware, there is only a limited amount of research in this area.

Congestion Due to the importance of IP networks, KNXnet/IP gateways are a key part for home automation systems like KNX. However, modern Ethernet networks operating at 100 Mbit/s or even 1 Gbit/s are orders of magnitude faster than KNX.

Neugschwandtner and Kastner studied the performance disparity between the two networks and suggest certain improvements [28]. As an offline measure, they propose to define the behavior and minimal traffic requirements of typical use cases (e.g. motion detection, brightness metering) to facilitate network planning. At runtime, they suggest that IP devices generating a traffic burst to different destinations (e.g. parallel read operations) should wait for a random time between the messages to reduce congestion on the KNXnet/IP gateway.

A practical analysis based on measurements is presented by Cavalieri [29]. His measurements confirm that the KNXnet/IP gateway even poses a problem under "traffic conditions not particularly critical" and telegrams get dropped. He suggests using a priority/FIFO queuing inside the KNXnet/IP gateway to reduce the loss of high and medium priority telegrams.

2.4.4 IEEE 802.15.4 and Zigbee network measurement

Most Zigbee stacks are proprietary and bundled with the chip manufacturer's hardware. On the other hand, IEEE 802.15.4 is an open standard and supported by various operating systems for embedded devices, for example TinyOS[1] or Contiki (Subsection 4.2.2). We could not find material related to topology discovery on IEEE 802.15.4 which is not surprising as the topology is maintained by higher layers, for example ZigBee.

Performance analysis Ullo and Velotto conducted a simulation based analysis of a wireless sensor network based on the 802.15.4 protocol [30]. They observed and assessed parameters like throughput, latency and service degradation. They reach to the conclusion that 802.15.4 networks are suitable for smart grid applications. However, they do not recommend IEEE 802.15.4 based networks for high performance and time critical applications within smart grids.

Multi-sniffer To analyze wireless networks, a multi-sniffer system can be beneficial due to the limited radio range of nodes. Yu Yang et al. present a multi-sniffer system called "Sensor Network Analysis and Management Platform" (SNAMP) where they collect data gathered from multiple wireless sensor node and combine this information into one application [31]. They claim that visualization is a key aspect to understanding a network. However, the authors failed to state which kind of network they are analyzing.

A similar approach is taken by the authors of "Sensor Network Distributed Sniffer" (SNDS) analyzing IEEE 802.15.4 based networks. The authors focus on large amounts of traffic and time synchronization between the sensor nodes [32]. They report good results in terms of stability, time synchronization and protocol analysis.

A multi-sniffer approach could be beneficial in our case as well to extend the radio range. Furthermore, we could start to exchange traffic between our two measurement nodes, for example to implement bandwidth measurement (Subsection 8.1.2).

Security framework Killerbee is an open source project providing a "Framework and tools for exploiting Zigbee and IEEE 802.15.4 networks" [33]. It contains functionality for sniffing, packet injection (enabling replay attacks), active and passive scanning for Zigbee devices, etc. One drawback of this framework is that the hardware support is limited to a few selected 802.15.4 USB sticks.

2.4.5 Distributed smart grid control

A smart grid is inherently a distributed system making it more resilient against malfunctions, facilitating fast reaction time and supporting microgrid islanding mode. Therefore, it is not surprising that the design ideas for smart grid control infrastructures are designed as distributed systems as well.

[1] `http://www.tinyos.net/`

Pipattanasomporn et al. propose a multi-agent system using TCP/IP for communication [34]. The purpose of the system is to control and monitor the smart grid, for example when detecting a contingency situation, the control agent sends a message to the circuit breaker. In their simulations, they show that a multi-agent system has the capability to disconnect and stabilize their simulated microgrid in case of a power outage.

For the ABB smart grid demo lab, a multi-agent system has been implemented as well [35]. Each agent handles a group of appliances belonging together, for example one agent takes care of the KNX building automation system. The agents are interconnected by an IP network and can subscribe to events from other agents. The database agent for example subscribes to all events, creating a history. A domain-specific language called smartScript makes use of the agent infrastructure to provide a high level language allowing users to interact with all systems in a homogeneous and intuitive way.

2.5 Related software

In this subsection, we review software products implementing network measurement functionality for either KNX or IEEE 802.15.4 based networks.

2.5.1 KNX

Calimero The Automation Systems Group from the Vienna University of Technology in Austria performs research in the area of building automation. Most notably, they developed a KNX library for Java named Calimero which provides KNX bus access using a KNXnet/IP gateway [36]. Dominik Windhab used this library in his bachelor thesis to develop a control for KNX devices based on Windows mobile [37]. The implementation of our KNX probe is based on Calimero as well (Section 5).

ETS ETS is the official configuration, network deployment and project management tool offered by the KNX consortium. Due to the standardization of commands on the application layer, it is manufacturer independent. A KNX deployment is stored as a project containing all deployed devices, their settings and properties (Figure 5). Apart from project management, ETS offers some network analysis and debugging functionality:

- Sniffing: ETS supports the bus monitoring and group monitoring mode. The sniffer shows not only the raw data but also interprets it with respect to the database entries (e.g. the destination address is resolved to the actual device name).
- Subnet device discovery: discover all devices within a certain subnetwork. The results can be linked with the project to see if there are discrepancies.
- Device confirmation: scan single device addresses to see whether this device exists or not.

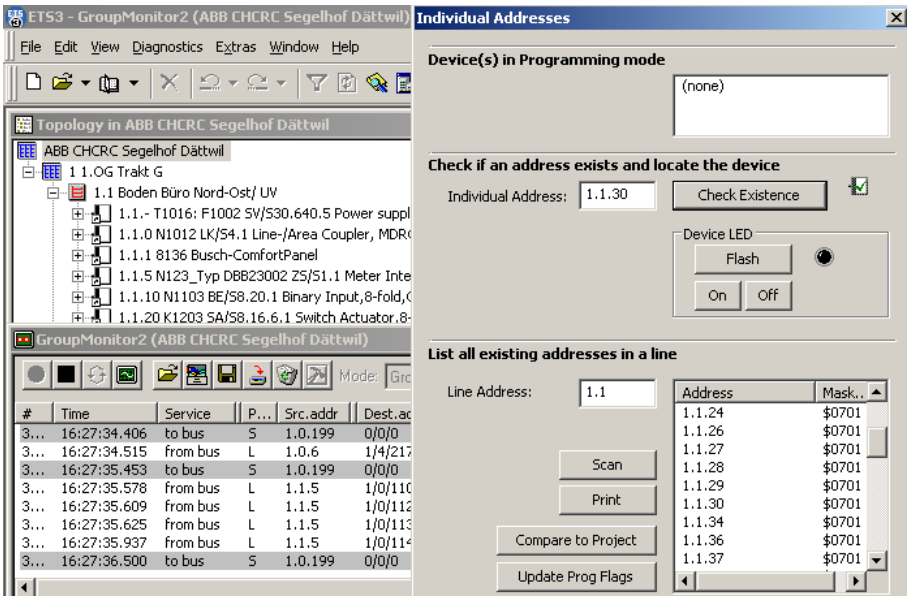
ETS3 - GroupMonitor2 (ABB CHCRC Segelhof Dättwil)
File Edit View Diagnostics Extras Window Help
Topology in ABB CHCRC Segelhof Dättwil
ABB CHCRC Segelhof Dättwil
1 1.OG Trakt G
1.1 Boden Büro Nord-Ost/ UV
1.1.- T1016: F1002 SV/S30.640.5 Power suppl
1.1.0 N1012 LK/S4.1 Line-/Area Coupler, MDR
1.1.1 8136 Busch-ComfortPanel
1.1.5 N123_Typ DBB23002 ZS/S1.1 Meter Inte
1.1.10 N1103 BE/S8.20.1 Binary Input,8-Fold,
1.1.20 K1203 SA/S8.16.6.1 Switch Actuator.8-
GroupMonitor2 (ABB CHCRC Segelhof Dättwil)
Mode:
Time Service P... Src.addr Dest.ac
3... 16:27:34.406 to bus S 1.0.199 0/0/0
3... 16:27:34.515 from bus L 1.0.6 1/4/217
3... 16:27:35.453 to bus S 1.0.199 0/0/0
3... 16:27:35.578 from bus L 1.1.5 1/0/110
3... 16:27:35.609 from bus L 1.1.5 1/0/112
3... 16:27:35.625 from bus L 1.1.5 1/0/113
3... 16:27:35.937 from bus L 1.1.5 1/0/114
3... 16:27:36.500 to bus S 1.0.199 0/0/0
Individual Addresses
Device(s) in Programming mode
(none)
Check if an address exists and locate the device
Individual Address: 1.1.30
Check Existence
Device LED
Flash
On
Off
List all existing addresses in a line
Line Address: 1.1
Scan
Print
Compare to Project
Update Prog Flags
Address Mask..
1.1.24 $0701
1.1.26 $0701
1.1.27 $0701
1.1.28 $0701
1.1.29 $0701
1.1.30 $0701
1.1.34 $0701
1.1.36 $0701
1.1.37 $0701

Figure 5: ETS - the official KNX project management tool. The top left subwindow shows the database information of the current deployment. Below, the group monitor displays the sniffed traffic. The right side window contains the network analysis functionality.

ETS can be connected to the KNX bus through a serial cable, USB or KNXnet/IP (given that the appropriate KNX device is installed). It does not offer measurement functionality, for example to measure the RTT.

ABB i-bus tool The ABB i-bus tool is a stand-alone tool to control appliances and the main focus is to present the end device functionality to the user [38]. It can be extended by plugins to add support for additional device types. The ABB i-bus tool accesses the KNX bus either through USB, serial port or KNXnet/IP. It does not offer network measurement functionality.

Wireshark plugin When a third-party application is connected to a KNXnet/IP gateway, Wireshark can intercept the KNXnet/IP UDP packets and decode their content. For Wireshark to understand the KNXnet/IP protocol details, a plugin named *KNXnet/IP Wireshark dissector* has to be installed [39]. However, the cEMI payload is not decoded and hence the KNX addresses are not visible.

Net'n Node Developer This stand-alone tool is designed by Weinzierl Engineering GmbH in Germany and intended for development, debugging and testing [40]. Weinzierl sells other components for KNX device development and programming, for example a KNX TP 1 communication stack.

2.5.2 IEEE 802.15.4

Sensor Network Analyzer The Sensor Network Analyzer (SNA) software by DaintreeNetworks is a comprehensive tool for analyzing 802.15.4 based networks [41]. It offers packet sniffing and header field decoding for 802.15.4 and Zigbee, network visualization by actively or passively scanning for devices and measurements like throughput, latency or retransmission ratios. The software is closed source and is coupled with a network adapter from Daintree. Production of hardware and software has been discontinued since March 2010.

Development kits Producers of 802.15.4 chips often sell development boards and provide free software usable with their products. For example, the smartrf packet sniffer is a PC application usable in combination with system on chip manufactured by Texas Instruments [42]. It runs on Windows and offers a GUI showing the captured packets in decoded form. It furthermore offers a feature encapsulating all sniffed traffic into UDP packets making it usable for other applications.

Z-monitor Z-monitor is an open-source tool offering frame decoding and protocol analysis for 802.15.4 networks. It supports decoding ZigBee, 6LoWPAN and RPL (routing protocol for low power and lossy networks). It is not coupled with specific hardware and requires input from a sensor mote, for example running TinyOS or Contiki.

Wireshark The open-source tool Wireshark is widely used to analyze IP based traffic from Ethernet and 802.11 based networks. However, it is also capable of decoding 802.15.4, Zigbee and 6LoWPAN traffic. There are various ways to gather the input data for Wireshark, but they are less convenient compared to sniffing on an Ethernet network interface offered by the host OS.

The principle of obtaining input is similar to Z-monitor: we need a sensor mote providing the input in a Wireshark compatible format. The Wireshark Wiki suggests using a Exegin Q51 IEEE/802.15.4 ZigBee Transceiver which encapsulates the sniffed traffic into TCP/IP packets.

For this project, we used Wireshark to familiarize ourselves with the network and for testing and debugging. Using our Econotag hardware platform, we provide the input to Wireshark through the command line (Subsection 4.2.1).

3 Project overview

This chapter provides an overview over the ABB smart grid demo lab and the architecture and functionality of our software. Afterwards, a set of software modules is introduced which are shared between the application and the control units (components in charge of network measurement). Finally, we present the basic design principle of our control units.

3.1 ABB smart grid demo lab

The ABB smart grid demo lab is a research infrastructure deployed at ABB Corporate Research Center in Baden-Dättwil, Switzerland[2]. It consists of typical smart grid components manufactured by ABB as well as third-parties and is aimed towards research and development [35]. The smart grid demo lab network interconnects for example photo-voltaic panels, smart meters etc. using a multitude of different network technologies like Ethernet, KNX or Zigbee.

For our research, we are interested in the network technology interconnecting these components, namely 802.15.4 and KNX. In contrast to control platform development, we are only concerned with the network protocols and the performance, but not the semantics of the commands. As we focus only on the underlying network protocol, our research is applicable in areas other than smart grids as well.

3.1.1 Network structure

All smart grid devices are indirectly connected to a LAN, some of them (solar panels and the car charging pole) indirectly via a smart meter (Figure 6). This IP network is a standard 100 Mbit/s Ethernet LAN, physically separated from the company network. We focus our analysis of KNX and 802.15.4 ignoring the other components and the network of agents (Subsection 2.4.5). However, by extending our work to cover IP-based networks as well, we would be able to capture the entire smart grid demo lab network (Subsection 8.1.2).

The KNX bus is connected to the IP network using a KNXnet/IP gateway. The bus has a topology itself consisting of a three level hierarchy not illustrated in the figure (Subsection 2.1.2).

The right side of Figure 6 shows two special power plugs named *Plugwise Circle*[3]. These special power plugs (named Plugwise hereafter) have integrated monitoring functionality and accept remote commands to turn the attached devices on and off. The Plugwise communicate with a manufacturer specific USB dongle using Zigbee. The communication is encrypted and the protocol between them is proprietary. The USB dongle is attached to a PC which serves as relay between the IP network and the Plugwise.

[2] `http://www.abb.com/cawp/abbzh254/ec72bb280fd24d47c1256b5700522f3a.aspx`
[3] `http://www.plugwise.com`

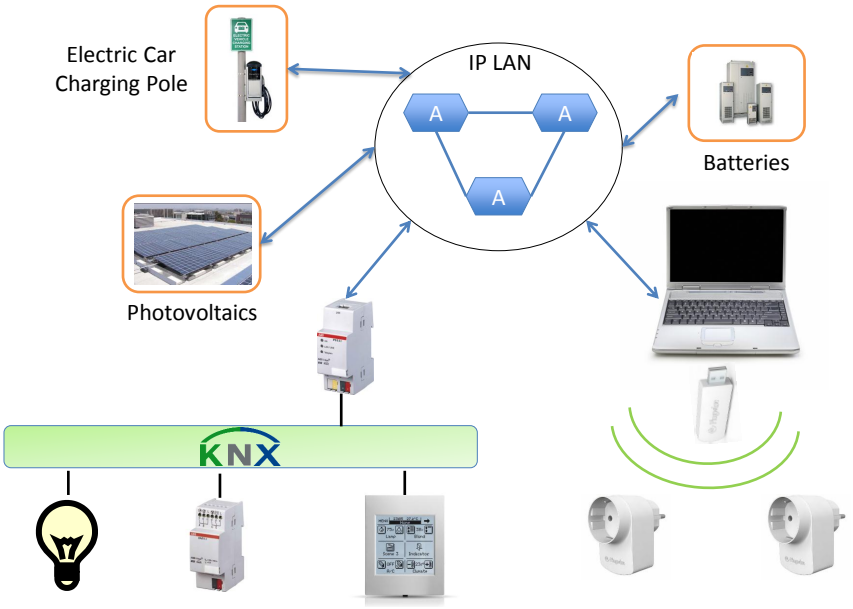

Figure 6: High level network overview. All smart grid demo lab devices are interconnected by an IP network. KNX is connected to the LAN using a KNXnet/IP gateway whereas the Plugwise power plugs communicate with the LAN by their associated USB dongle and an intermediate computer.

3.2 Architecture

This subsection provides an overview of our system. We start by motivating our design choice and then present the architecture including the terminology for the different parts of our system.

3.2.1 Motivation

The cornerstone of our design is the distributed architecture with direct access to the NUT. First, we motivate the assumption of an uncooperative network. Afterwards, the arguments towards direct, external network access are presented. In the end, we discuss the advantage of a distributed system with respect to the distance between different networks.

Uncooperative The approach of implementing measurement functionality into smart grid network components is well-established (Subsection 2.4.2). However, we believe that this approach is too restrictive in the future due to the following reasons:

- A microgrid may involve several buildings or an entire neighborhood. It is not beneficial for the breakthrough of smart grid technology if all

participants are forced to buy products from one particular vendor (vendor lock-in).

- If a cooperative approach should work across vendors, they would have to agree on a standard. Standardization is known to be complicated and time consuming.

To keep our measurement approach universal, we assume the nodes to be uncooperative.

Direct network access In a heterogeneous network environment, some network types may only be reachable through a custom, high level protocol. Often, this data is aggregated and the end devices are not directly accessible. The above mentioned agent network running as part of the ABB smart grid demo lab is a good example: it presents the functionality to the user without requiring knowledge about the underlying network structure and the protocols. However, for our purpose, it is crucial that we have direct access to the nodes because we want to measure their connectivity and network properties.

Even if we would find a way to reuse such an existing, high level protocol for measurement purposes, other problems arise. The biggest issue is reusability: our software would be tailored towards one custom, most likely not standardized, protocol. Another issue is taking active measurements which requires sending data to the nodes. The measurement infrastructure should not interfere with normal operation and finding a high level command without side effects may be difficult.

External When deploying a manufacturer-independent measurement infrastructure, we cannot assume that we have the same network access as their devices. For example, we may not know the encryption keys or the exact routing algorithm. Furthermore, if network membership is managed on higher layers (in our case ZigBee with respect to IEEE 802.15.4), we cannot even join the network.

The common denominator is the network specification of the underlying network. Hence, we decided to choose an external measurement approach.

Distance In a microgrid, the various networks are distributed among one or several buildings. Hence, it is impossible to setup just one machine and equip it with different network interfaces to gather the measurements - it requires a distributed approach. Furthermore, the closer we are to the actual NUT (in terms of other devices in between), the less interference (e.g. queuing in switches) and the more precise our measurements.

3.2.2 Structure

To meet the demands we just described, Smart Eagle is comprised of multiple, stand-alone software components which can either run on the same or different machines. The base architecture is shown in Figure 7.

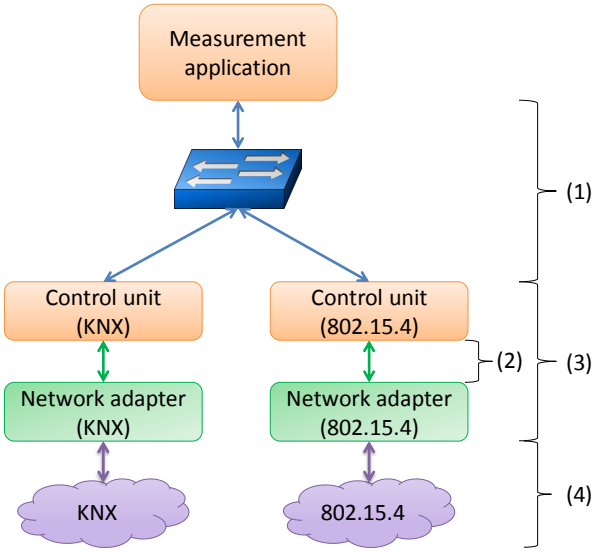

Figure 7: Smart Eagle architecture overview. Legend: (1) intermediate network, (2) interconnect link, (3) probe and (4) NUT.

Measurement application This is the top level application which controls the entire system, stores all the measurements gathered from the probes and interacts with the user. As the measurement application stores the entire dataset, it is not intended to run on embedded systems (in contrast to the probe).

Control unit Each type of network has its associated *control unit*, which we can think of as an intelligent relay doing format conversion and some basic preprocessing. It is connected to the network adapter by a link we refer to as *interconnect link*. The type of interconnect depends on the NUT (e.g. USB, serial, Ethernet) and the protocol can either be predetermined by the manufacturer (in our case KNX) or chosen by the control unit programmer (in our example 802.15.4). This depends on whether the network adapter is freely programmable or not.

Network adapter The *network adapter* provides access to the NUT. Depending on the type of network, this component may be a programmable piece of hardware or just an interface providing access to the network.

Probe The bundle of network adapter and control unit is called a *probe*. A probe is network specific and exchanges data with the measurement application using an *intermediate network*. The intermediate network is usually a fast (compared to the NUT), common network type.

Functionality	IEEE 802.15.4	KNX
Link layer RTT	✓	✓
Application layer RTT	×	✓
Subnet sweep	✓	✓
Topology discovery	×	✓
Channel scan	✓	×
Sniffing	✓	✓
Monitoring	✓	✓
Network adapter configuration	✓	×
Link quality	✓	×

Table 3: Overview of functionality provided by Smart Eagle for the two different network types. Legend: ✓: available and implemented, ×: not applicable.

3.3 Application functionality

This subsection introduces the Smart Eagle measurement application providing a quick overview about its functionality. The GUI structure and the measurement functionality is explained later in more detail.

3.3.1 Measurement functionality

The available measurement functionality is summarized in Table 3. Application layer RTT measurements are not available for IEEE 802.15.4 networks because the specifications are only up to the link layer. KNX offers no channel scan because it is wired network with one channel only and the KNX network adapter cannot be configured.

3.3.2 Graph view

For KNX, a graph view has been implemented visualizing the topology (Figure 8). Because the graph gets wide, the user can select whether to show the nodes in the subnetwork or not. The orange nodes indicate parent nodes having children. This graph feature is not available for IEEE 802.15.4 based networks.

3.4 Shared modules

This subsection describes a set of modules which are used in the measurement application as well as in the control units. We start by explaining our logging facility and then discuss the locator beacon service which allows the measurement application to automatically find the control units.

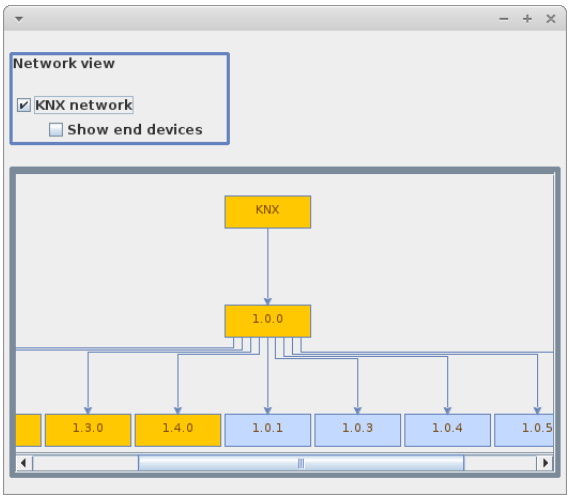

Figure 8: KNX graph view showing the network topology. The orange colored nodes have children. When selecting the *show end devices* check box, level three in the hierarchy is displayed as well.

3.4.1 Logging facility

There are several distributed logging facilities available on the market, an open source example is Log4j by the Apache foundation[4]. We decided to implement our own solution for several reasons:

- The communication infrastructure between measurement application and control unit is already available. We can reuse this without adding additional complexity to our application. In fact, the process of providing log data and sniffer data is analogous.
- Our application only requires simple logging functionality. We merely distinguish between three categories of logs: debug, info and error.
- Having all the logs in a simple ring buffer allows easy integration into the GUI.

The logger service can be accessed in the application through a set of static functions.

3.4.2 Locator beacon

Manually managing a distributed system is cumbersome and decreases the customer acceptance. Especially for large systems, maintainability and self-configuration are essential features, illustrated for example by DHCP.

[4]http://logging.apache.org/log4j/1.2/

```
1 route add default gw <gateway IP> eth0
```

Listing 1: Manually adding a default route to the Linux routing table. The *<gateway IP>* is the IP address of the default gateway we want to add.

In our system, the control units advertise themselves by UDP multicast. As soon as they are started, they send periodically a beacon containing their serial number and network type for which they are responsible. We introduced identification numbers to uniquely identify the networks, otherwise we would be unable to distinguish for example multiple KNX networks. The measurement application receives the beacon and extracts the network type, identification number and source IP.

We chose the control units as beacon sender to allow multiple measurement applications running simultaneously. However, this is currently not supported as only one measurement application can fetch the sniffer data and logs.

Next, we describe an issue regarding a missing default route which we encountered while using Java UDP multicast. Afterwards, we discuss how the beacon receiver ensures that it only notifies higher layer once upon receiving multiple beacons.

Default route We use Java multicast sockets to send an receive the UDP multicast beacons. The setup involves two steps: first, we open a multicast socket on a certain port and afterwards, we join the multicast group.

In our test network, the second step failed with an *IO Exception* because the DHCP server in the smart grid demo lab IP network did not provide a default route which seems to be required by the Java group join operation. The default route enables Internet Group Management Protocol (IGMP) packets to be sent to routers allowing a multicast network across subnetworks [43]. As a workaround, the default route needs to be added manually (Listing 1).

Single notification An application relying on the beacon service (client) should only get one notification about a newly detected device, no matter whether multiple beacons were received. Furthermore, multiple different beacons could arrive simultaneously requiring a queuing mechanism. We resolved this issue using two queues:

- Discovered: control units which newly registered (inserted only once).
- Connected: control units which are already handled by the client.

When a new beacon arrives and is not found in any of these queues, it is inserted into the discovered queue. The client blocking on the discovered queue gets automatically notified, processes the new control unit and marks it as handled which removes it from the discovered queue and adds it to the connected queue. If the connection to a control unit is lost, the client notifies the beacon service which removes it from the connected queue.

3.5 Control unit architecture

In this subsection, we explain the base architecture for communication between the measurement application and the control units. It is based on a client - server approach: the measurement application makes a request to the control unit and the control unit responds.

We start by motivating our choice to use a webserver and XML for communication. Afterwards, we discuss fault handling and the resulting transaction executor framework. In the end, we present an extension to the transaction executor framework to deal with concurrency and locking.

3.5.1 Communication

Communication between the control units and the measurement application takes place over an IP network (intermediate network). As a first approach we used Java TCP sockets to transmit string messages. However, we soon realized that getting the multithreading and queuing right (due to the blocking Java sockets) is cumbersome, error-prone and time consuming. We decided to use a webserver instead. In the next paragraph, we present the advantages of this approach.

Web interface There are various web server libraries freely available, for example Apache Tomcat[5] or Jetty[6]. We decided to use Jetty because it is advertised as light weight and free for both commercial and non-commercial use. Jetty implements the servlet API providing a convenient way for client-server communication and is widely used in combination with HTTP to dynamically generate websites. The response is transmitted in XML format. Using the concept of a webserver in combination with XML has several advantages:

- It relieves the programmer from dealing with sockets, thread management and concurrency issues. Each request is handled by a separate thread and the amount of threads in the system is administered by Jetty (load dependent).
- HTTP allows to test and debug the control unit using a web browser. Furthermore, sending HTTP requests from inside applications is widely used, simple and well understood.
- XML is a standardized format, programming language independent and natively supported by Java SE through JAXB[7]. The XML is generated directly from an instance of a Java class containing fields having JAXB annotations. When unmarshalling the XML, we get back an object from the corresponding type carrying the data from the XML.

[5] http://tomcat.apache.org/
[6] http://jetty.codehaus.org/jetty/
[7] http://jaxb.java.net/

- Fault handling (e.g. invalid request, unexpected connection termination) are already addressed by the webserver as these are common problems in web-based applications.

We increased robustness and simplified fault handling even more by keeping our server stateless. Fetching passively obtained data (sniffer, logs) is polling based. Executing a measurement and fetching its result is only one request as well.

On top of Jetty, we implemented a framework treating measurements as transactions. In the next subsection, we motivate the choice of transactions and present our implementation.

3.5.2 Transactions

Fault handling is of major concern when dealing with network measurements: they can fail in various ways, e.g. failure of intermediate network, sudden termination of network connection, unexpected response from network adapter, etc.

Apart from failure recovery, this raises other questions: what is the semantics of a partially completed measurement? What can we say about a subnet if device discovery crashed before the discovery operation was completed?

From our point of view, it does not make sense to show the user partial data as this is more confusing than helpful. Furthermore, we anticipate that the correct handling of partial data (which should happen seldom) requires a considerable additional effort when developing the measurement application.

To simplify and clarify the reasoning about measurement requests, we treat all queries to the control units (including measurements) as a transaction: either the request completed successfully or it failed. The measurement application is provided with a clear semantic and does not have to worry about the state of the control unit.

ACID The notion of a transaction is often used in combination with the ACID properties. They are only partially applicable in our case of measurement transactions:

- *Atomicity* is our main focus as it presents a simple and clearly understandable semantic.

- *Consistency* is not an issue for measurement requests as the control unit does not store the response data. However, sniffer data is buffered on the control unit and fetching it can fail. While fetching, we clear the buffer and keep our own reference to the data structure to avoid concurrency issues with JAXB. Strictly speaking, we could just throw away the buffered data in case of a failure and the control unit would still be in a valid state. Yet, we implemented a rollback to write back the data into the buffer if the fetch operation fails.

- *Isolation* is provided through the multithreading behavior of the webserver (each measurement runs in its own thread) and the measurements are independent of each other.
- *Durability* does not apply as the control units do not store data permanently.

Next, we discuss how transactions could be mapped to HTTP status codes in an ideal case and how we implemented them.

HTTP status codes Ideally, transactions could be represented by HTTP status codes in the following way: a "202 Accept" is sent when the query arrives, a "200 OK" if the transaction is completed or a "500 Internal Error" in case of an abort. In the event of an invalid query, the control unit would respond with "400 Bad Request".

Unfortunately, it cannot be implemented like this as only one HTTP status code can be sent per request. For our implementation of the control units, we therefore omit the "202 Accept". This is a drawback in case of long running queries as the client has no way of knowing whether the query as arrived or not. A work around would be to send the "202 Accept" as part of the HTTP body.

To simplify the programmers task when dealing with transactions, we implemented a framework which deals with HTTP status codes and fault handling. The implementation of this framework is presented in the next paragraph.

Transaction executor We developed a transaction framework to simplify the implementation of transaction based measurements. Such systems are commonly known as Transaction Processing Monitors (TP Monitors) but we stick to the term transaction executor (TE) as it is more appropriate in our case.

Our *WebTransactionExecutor* (WTE) encapsulates the actual measurement functions and deals with a wide range of exceptions. Each measurement is required to implement our *WebTransaction* interface and provides a set of functions to the WTE (Table 4). The WTE is specially designed to handle HTTP request as it forces the measurement function to decode the HTTP parameters and verify their validity.

If the measurement function runs into an exception, it can either handle the exception and return a valid measurement or throw an *InternalMeasurementException* which is treated by the WTE as an abort. The most common exceptions like JAXBException or IOException are directly handled by the WTE, i.e. the measurement function terminates with an abort.

We extended the TE with a component to help the developer dealing with concurrency. The issue of concurrency and the extension to the TE are discussed in the next paragraph.

Concurrency A major source of concurrency are parallel requests from the measurement application to the control unit. It is often convenient to control concurrency before the actual measurement started as it reliefs the back-end implementing the low level functionality from dealing with nasty concurrency

Function	Purpose
argumentsValid	Decode and verify the HTTP query string. If the arguments are invalid, the TE aborts with HTTP status code “400 Bad Request”.
execute	The actual measurement to be executed. It is up to this measurement function to return a valid response to the query (XML body).
rollback	Executed in case the transaction failed.
cleanup	Operations which should always be executed, even in case of an abort.

Table 4: Functions required by the WebTransaction interface. Each measurement implementing this interface can be passed to the TE for execution.

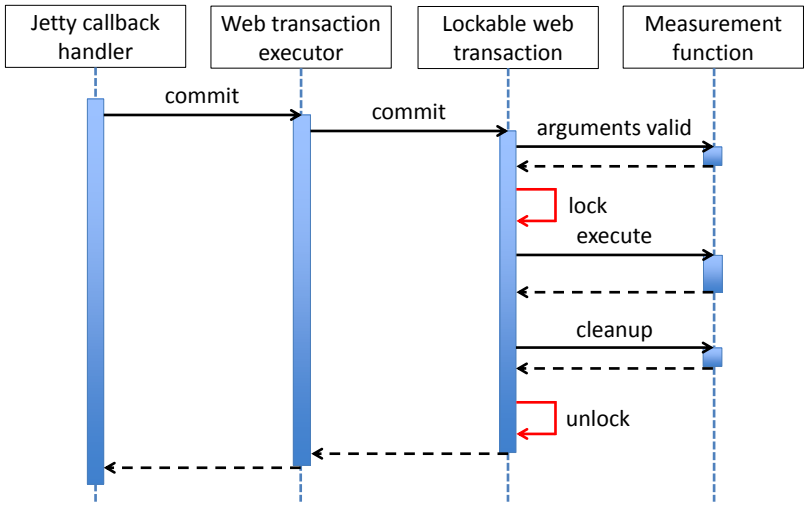

Figure 9: Execution of a measurement transaction with locking. The Jetty callback handler is invoked upon the arrival of a HTTP GET request. The WTE executes the measurement indirectly via a LockableWebTransaction.

issues. An example of such a concurrency issue is mapping the response from the network adapter to the corresponding request.

Concurrency control can can be implemented at different levels of granularity. Fine grained locking (e.g. prevent two measurements for the same address) requires knowledge about the NUT and cannot be implemented as part of the TE. Yet, coarse grained locking functionality like executing one transaction at a time or preventing only certain transaction types from being executed together is part of our TE.

To run a measurement with concurrency control enabled, the object holding the measurement function is encapsulated by a *LockableWebTransaction* taking a Java Reentrant Lock as argument. The LockableWebTransaction implements the WebTransaction interface and can be passed to the WTE which executes it (Figure 9). The reentrant lock is configured to be fair by enforcing that the measurement queries are processed in FIFO order.

As an alternative, we considered a request queuing system. However, this would have been more difficult to implement because it requires dealing with threading and wait-notify mechanisms.

4 IEEE 802.15.4 probe

In this section, we present the implementation of the IEEE 802.15.4 probe and we start by discussing the architecture. Afterwards, we present the implementation details for the network adapter including our hardware platform, the OS and the network measurement functionality. Next, we describe the interconnect between the network adapter and the control unit. In the end, the implementation of the control unit is presented.

4.1 Architecture

The 802.15.4 probe consists of a freely programmable network adapter which is attached to a PC running Linux and hosting the control unit (Figure 10). The network adapter is connected to the PC through USB but registers as a serial console. The control unit is a Java application providing an interface between the network adapter and the measurement application. In addition, it deploys and launches the Contiki OS on the network adapter.

Network layer Initially, our idea was to analyze Zigbee networks. However, we decided to go one layer below for the following reasons:

- Encryption: 802.15.4 offers link layer encryption making it impossible for us to interact with higher layer protocols.
- Generality: by limiting ourselves to the link layer, we can evaluate all sorts of traffic based on IEEE 802.15.4 (e.g. Zigbee, 6LoWpan, etc.).
- Zigbee stack: there is no well tested Zigbee stack available for our platform

As a consequence, our 802.15.4 probe can only operate within the range of the 802.15.4 radio as network membership and routing is left to higher layer protocols. The current implementation is capable of monitoring the traffic and offers functionality for device discovery and RTT measurement.

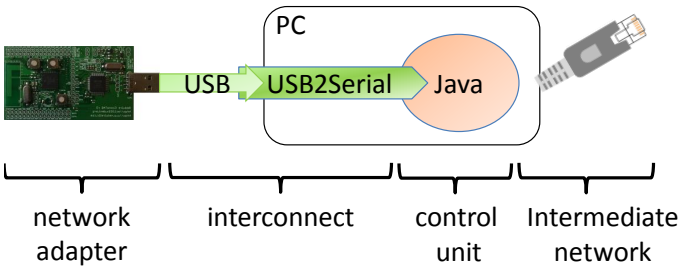

Figure 10: Design and interfaces of the 802.15.4 probe. The network adapter is attached by USB but appears as serial console to the control unit.

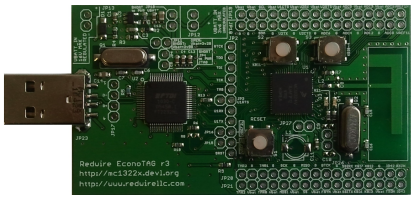

Figure 11: Econotag development board manufactured by Redwire. The 802.15.4 wireless interface is integrated into the Freescale system-on-a-chip.

4.2 Network adapter

All measurements are taken on the network adapter which means that the control unit only sends a command to the network adapter and waits for one or multiple responses to arrive. This is important for measuring the RTT as a USB connection suffers from significant delay and delay variation [44].

In this subsection, we discuss our hardware platform and the OS running on top of it. Afterwards, we present the implementation of the various network measurement techniques.

4.2.1 Econotag hardware platform

To access 802.15.4 based networks, we use a development kit called *Econotag* manufactured by Redwire[8] (Figure 11). This board has a Freescale MC13224v ARM7 microcontroller with 802.15.4 radio on board and provides two universal asynchronous receiver/transmitter (UART) interfaces.

We cannot use for example the Plugwise dongle, as the protocol between USB stick and host PC is proprietary. The USB dongle only emulates a serial interface to the host PC through which it accepts commands and sends data [45]. The 802.15.4 and Zigbee stack is implemented entirely on the USB stick preventing us from taking measurements.

Libmc1322x M. Alvira (Redwire LLC) provides some scripts (e.g. for deployment), a set of demo applications (serial console output, 802.15.4 sniffing, etc.) and a C library for the MC13224v chip [46]. The C library facilitates convenient access to integrated chip functionality by an abstraction of the low level hardware functionality. The programmer can simply include a C header file to initialize the radio instead of dealing with hardware access using memory addresses.

Attaching Wireshark The Econotags can be used as an input source for Wireshark. For this, we need two ingredients: the *rftest-rx* program running on the Econotag to sniff the input and the *rftestrx2pcap.pl* script converting the output from the Econotag into a Wireshark compatible format. Both pieces are

[8] `http://www.redwirellc.com`

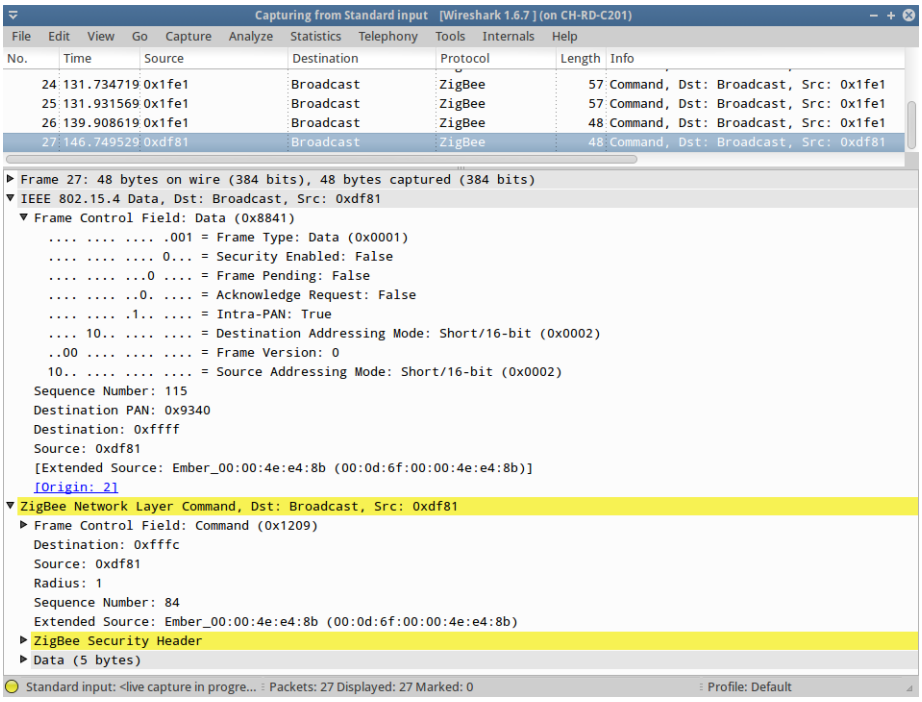
Capturing from Standard input [Wireshark 1.6.7] (on CH-RD-C201)
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
No. Time Source Destination Protocol Length Info
24 131.734719 0x1fe1 Broadcast ZigBee 57 Command, Dst: Broadcast, Src: 0x1fe1
25 131.931569 0x1fe1 Broadcast ZigBee 57 Command, Dst: Broadcast, Src: 0x1fe1
26 139.908619 0x1fe1 Broadcast ZigBee 48 Command, Dst: Broadcast, Src: 0x1fe1
27 146.749529 0xdf81 Broadcast ZigBee 48 Command, Dst: Broadcast, Src: 0xdf81
Frame 27: 48 bytes on wire (384 bits), 48 bytes captured (384 bits)
IEEE 802.15.4 Data, Dst: Broadcast, Src: 0xdf81
Frame Control Field: Data (0x8841)
.... .001 = Frame Type: Data (0x0001)
.... 0... = Security Enabled: False
....0 .... = Frame Pending: False
....0. .... = Acknowledge Request: False
.... .1.. .... = Intra-PAN: True
.... 10.. .... = Destination Addressing Mode: Short/16-bit (0x0002)
..00 .... = Frame Version: 0
10.. .... = Source Addressing Mode: Short/16-bit (0x0002)
Sequence Number: 115
Destination PAN: 0x9340
Destination: 0xffff
Source: 0xdf81
[Extended Source: Ember_00:00:4e:e4:8b (00:0d:6f:00:00:4e:e4:8b)]
[Origin: 21]
ZigBee Network Layer Command, Dst: Broadcast, Src: 0xdf81
Frame Control Field: Command (0x1209)
Destination: 0xfffc
Source: 0xdf81
Radius: 1
Sequence Number: 84
Extended Source: Ember_00:00:4e:e4:8b (00:0d:6f:00:00:4e:e4:8b)
ZigBee Security Header
Data (5 bytes)
Standard input: <live capture in progre... Packets: 27 Displayed: 27 Marked: 0 Profile: Default

Figure 12: Plugwise wireless communication captured by the Econotags displayed in Wireshark. The frame decoding shows the addressing mode and encryption on the Zigbee layer but not on link layer.

included in the libmc1322x package. To get the Wireshark sniffer running, the following steps are carried out:

1. Deployment of the rftest-rx program on the Econotag (Listing 2, Line 1). Upon success, we see the raw sniffer output appearing on the serial console of our PC.

2. Before attaching Wireshark to the serial console, we may need to change the radio channel. We do this by writing newline characters on the serial console. For each newline character, the channel number is incremented by one.

```
1 mc1322x-load.pl -f rftest-rx_redbee-econotag.bin -t /dev/ttyUSB2
                                    -c 'bbmc -l redbee-econotag reset'
2 libmc1322x/tools/rftestrx2pcap.pl -t /dev/ttyUSB1 | wireshark -k -i -
```

Listing 2: Forwarding the sniffer output from rftest-rx on the Econotag to Wireshark. The raw serial console output needs to be converted into a Wireshark compatible format.

3. Finally, we attach the conversion script to serial console and pipe the output to Wireshark (Listing 2, Line 2).

The result is an inexpensive yet powerful 802.15.4 sniffer capable of decoding Zigbee and 6LoWPAN as well (Figure 12). The frame shown in the figure is originated from a Plugwise. From the 802.15.4 frame control field, we see that no link layer encryption is active, communication is within the same PAN and that the devices use short addressing. The Zigbee layer is highlighted yellow because the Zigbee payload is encrypted. This protects (possibly sensitive) user data and the proprietary protocol making it impossible for other Zigbee devices to participate in the communication without knowing the secret key.

We use the Econotag hardware platform and Contiki as OS to perform our measurements. In the next subsection, we describe the setup, deployment and network stack of Contiki. In the end, we list the our modifications to the original Contiki code in order to execute our measurements.

4.2.2 Contiki

Contiki [9] is an open-source OS for embedded devices written in C. It is designed for small embedded devices and hence has low memory requirements (less than 10 Kb RAM and 30 Kb ROM for Contiki including IPv6 networking). It runs well on the Econotag platform, has a 802.15.4 stack and implements 6LoWPAN, IPv6, TCP/IP, etc.

Cross compiler As the Econotags have an ARM CPU on board, we need am ARM little endian cross compiler. The GNU Compiler Collection (GCC) is freely available for ARM in a precompiled version from Mentor Graphics (formerly Codesourcery) [47]. The compiler (arm-none-eabi-gcc) is installed by unpacking the archive and adding the binaries to the global search path. If the host machine is running a 64-bit Linux, the ia32-libs need to be installed in addition.

Build system The Contiki build system is based on GNU Make. For the user to compile the OS, the Makefile within the application directory has to be executed (Listing 3, Lines 1-3). All required components for the OS are included through other makefiles located throughout the entire Contiki directory structure. If the application consists of multiple files, they need to be appended to the

[9] `http://www.contiki-os.org/`

```
1 cd contiki/examples/hello-world
2 make TARGET=redbee-econotag BUILD=debug
3 make TARGET=redbee-econotag BUILD=debug hello-world.elf
4 mc1322x-load.pl -f hello-world_redbee-econotag.bin -t /dev/ttyUSB1
                                    -c 'bbmc -l redbee-econotag reset'
```

Listing 3: Compiling and deploying the hello-world application on a Econotag.

PROJECT_SOURCEFILES variable. This holds as well for certain libraries like *assert* which are not used otherwise.

For a complete, clean rebuild it is not enough to just execute *make clean* as the cleanup command does not remove all parts of the application. For this task, we use our own script (*clean.sh*) which removes all binaries from the application directory forcing a complete rebuild.

Deployment The libmc1322x package includes a Perl script (*mc1322x-load.pl*) assisting in deploying the binary to the Econotag. Before the deployment starts, the device needs to be reset. This can be done manually by pressing the reset button or automatically using the *bbmc* tool which is part of the libmc1322x package as well. Deployment and reset can be conveniently spliced together (Listing 3, Line 4).

Network stack Each layer implements the same set of functions (for example: send_packet, packet_input) and stores a pointer to these functions in a struct. In the platform configuration file, these structs are assigned to constants representing the network stack (Listing 4, lines 1-4) . Within the network stack, higher and lower layer functions can be called using these constants (Listing 4, lines 6-7). The approach allows reconfiguring the network stack by simply assigning different structs to the constant.

To modify the frames and obtain the information we want, we do not always follow the layered approach provided by the network stack (Figure 13). The Smart Eagle process running on Contiki invokes the send function directly on the link layer and the frame is then modified within the network stack. The intended purpose of the unmodified components in the network stack are the following:

- *CSMA*: medium access protocol. Only transmit when the medium is free.
- *nullrcd*: radio duty cycling (RDC). Determines when the radio is turned off to save power.
- *Contiki maca* and *maca*: a general (contiki maca) and a platform specific (maca) send and receive function accessing the radio.

```
1 // Platform configuration file: contiki-conf.h
2 #define NETSTACK_CONF_MAC csma_driver
3 #define NETSTACK_CONF_RDC nullrdc_driver
4 [...]
5
6 // Usage inside network stack in send_packet of csma
7 NETSTACK_CONF_RDC.send(sent, ptr);
```

Listing 4: Contiki network stack configuration and usage. The define statements are located in the platform configuration file. The constants can be used throughout the system to call the network stack functions.

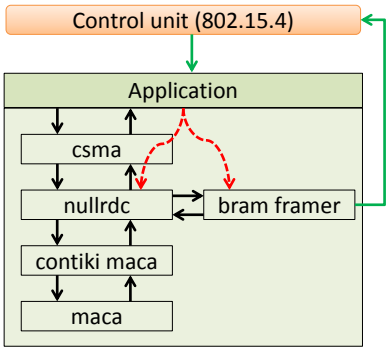

Figure 13: Network stack of the Contiki-based network adapter. Communication through the red arrows bypasses the layered network architecture. The green arrows represent the communication on the interconnect. Data frames from the application are directly passed to the link layer.

Next, we describe our modifications to Contiki in order to implement our measurement functionality.

Modifications We deliberately kept the amount of changed files in the network stack small to avoid cluttering up the code. Throughout the entire implementation on Contiki, the following files were added or significantly changed:

- examples/smart-eagle/smart-eagle.c
  Main process responsible for parsing and executing the commands received on the interconnect.
- examples/smart-eagle/sniffer-config.h
  Constants defining the commands used on the interconnect.
- examples/smart-eagle/parser.c
  Parsing functions for network addresses and commands received on the interconnect.
- platform/rebdee-econotag/contiki-conf.h
  The Contiki platform configuration file. Here, we configure the composition of the network stack.
- core/lib/bram_tools
  Routines for debug print and error handling.
- core/net/mac/bram_framer802154
  Implements sniffing and most of the functionality to set certain network packet parameters before sending the packet which are not accessible otherwise.

- core/net/mac/csma.c
  Functionality for RTT measurements: restart the timer for the RTT measurement before the packet is processed by the CSMA protocol.

- cpu/mc1322x/clock.c
  Increased clock precision.

- /cpu/mc1322x/lib/maca.c
  Disable ACK copy to avoid duplicate ACKs (Subsection 4.2.4).

Next, we discuss the implementation details and pitfalls when implementing the sniffing and measurement functionality on Contiki.

4.2.3 Sniffing

We select the nullrdc_driver to keep the radio always on because otherwise we would miss the traffic while the radio is asleep. To receive all traffic instead of only the traffic addressed to our device, the main application puts the radio into promiscuous mode. The frame parsing function is called in the beginning of the packet_input function in the nullrdc_driver and that is where we print out the packet (Figure 13).

To print the sniffed information to the serial console, we use the printf function. The sniffed information is packed into one long printf statement (instead of multiple smaller printf statements) to reduce the overhead.

4.2.4 Round trip time measurement

The basic idea is to send some data to the remote device while requesting a link layer ACK (this procedure is called an *ACK request*). Thereby we assume that a simple "0" does not have any side effects. The link layer does not check the payload but certain frame requirements have to be met for an ACK to be sent (Subsection 2.2.5). To allow proper configuration, we extended the framer with the following functions to modify the frame just before it is sent:

- set_pan_id: set source and destination PAN address. We do not support inter-PAN communication as our network adapter is not really part of any PAN.

- set_long_src_addr / set_short_src_addr: set either long or short 802.15.4 source address.

According to the IEEE 802.15.4 specifications, an *unexpected* source address (address filtering) is no reason for not sending an ACK. However, address filtering is a well-known firewall technique and may find its way into PANs as well. As we support setting the source address, address filtering could be circumvented by setting a valid source address (spoofing) which can be obtained conveniently through sniffing.

Instruction flow This paragraph briefly explains how a RTT measurement is performed. Afterwards, a few issues are discussed in more detail.

First, the control unit has to configure the network adapter by setting the correct PAN ID (for example obtained through sniffing) and (if required) a proper source address (red arrows in Figure 13). As soon as the settings are active, the new configuration applies to all outgoing traffic. This simplification is not a problem, as we only send probing traffic from the network adapter anyway.

Next, the control unit sends the ping command (RTT measurement). After receiving the command from the interconnect, the network adapter decodes the destination address, creates a packet with content "0" and sends the packet. Within the network stack, we extract and store the sequence number for later usage.

If the device responds, the link layer ACK is sniffed by the Econotag and passed through to the network stack. In the framer, we match the ACK sequence number against the previously stored sequence number. If it matches, we send a *pingack* to the control unit, otherwise a *snifack*.

Duplicate ACK Initially, every link layer acknowledgement we sniffed with Contiki was duplicated. However, our reference sniffer consisting of rftest-rx and Wireshark showed only one ACK.

We discovered that Contiki implements a feature which manually creates a copy of the ACK and passes it to higher layers in the network stack. This is useful in case hardware ACKs are enabled because otherwise higher layers would never see an ACK frame.

As discussed later (Subsection 4.2.5), hardware ACKs are disabled. Hence, both the original ACK and its copy are forwarded leading to duplicate ACKs. Disabling this feature using a C preprocessor macro resolved the problem.

Timer driver The Contiki timer driver is based on an integer which is incremented on each timer interrupt. The default interrupt frequency is chosen to be 100 interrupts per second and hence the timer variable accuracy is limited to 10 ms.

The RTT between two 802.15.4 devices is around 2.5 ms (value obtained experimentally) requiring a timer precision of at least 0.1 ms to get useful measurements. We adapted the interrupt frequency to get a precision of 0.08 ms. Having a clock frequency of 24 MHz and a prescaler (internal divisor of clock input frequency) of 128 leads to the following calculation:

$$\frac{24\,MHZ}{128} = 187500\,Hz \quad (1)$$

$$\frac{1}{187500\,Hz} = 5.3\,\mu sec \quad (2)$$

$$5.3\,\mu sec \cdot 15 = 80\,\mu sec \quad (3)$$

In Equation 1, we obtain the clock frequency after applying the prescaler which leads to minimum time of $5.3\,\mu sec$ between two consecutive interrupts

(Equation 2). We selected to get an interrupt every 15 ticks to get the desired timer resolution (Equation 3). Timer interrupt frequency and overhead is a tradeoff because the higher the interrupt frequency the more overhead we have. This timer configuration leads to a clock accuracy of $80\,\mu sec$ because we reset the timer to zero before each measurement. If we would subtract two timer values instead, the worst case measurement error would increase to $160\,\mu sec$.

4.2.5 Acknowledgements

In the last section, we explained that our RTT measurement is based on a data frame with the ACK request flag set. In this subsection, we discuss why the mapping of the frame and the ACK sequence number is performed in software (instead of hardware) and the implications. The sequence number mapping is required to assess to whom the frame belongs because ACK frames do not carry a source address (Subsection 2.2.5). We start by explaining the relationship between ACK and the CSMA/CA protocol.

CSMA/CA The IEEE 802.15.4 specifications define an unslotted CSMA/CA as medium access mechanism ([16], Subsection 5.5.4). It defines that an ACK (if requested) shall sent by the receiver within a certain time frame. The time frame is chosen in such a way that the ACK fits between the first data frame requiring an ACK and the second data frame.

Sending the ACK upon the reception of a data frame (receiver) and mapping the ACK to the data frame just sent (sender) is often implemented in hardware. The MC13224v implements *automatic acknowledgement reception* as part of their sequencer ([48], Subsection 9.5.1.2.1). This means that after sending a frame, the corresponding ACK is not passed to the CPU but processed in hardware. If the ACK is a response to the data frame we sent, the hardware responds with a *tx_success* and otherwise if a *tx_noack*.

Promiscuous mode For our project, we cannot rely on the automatic ACK feature because enabling the promiscuous mode disables auto-acknowledgement automatically ([48], Subsection 9.5.1.6). Other chips like the Atmel AVR2025 chip exhibit the same behavior ([49], Subsection 6.1.1.3). Thus we need to match the ACK to their corresponding data frames in software and therefore we need to address three elementary questions:

1. After sending an ACK request, how long shall the network adapter wait for a response?
2. Can we receive ACKs from other sources while we are waiting for our ACK to arrive?
3. How often does it happen that the ACK from our RTT measurement has the same sequence number as an ACK from another source?

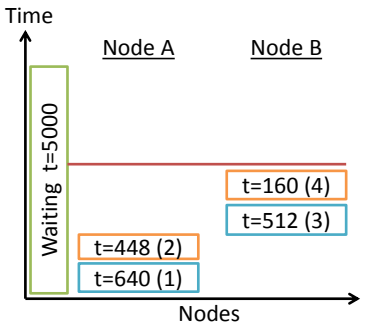

Figure 14: IEEE 802.15.4 network adapter receiving a foreign ACK (red line) because A sends an ACK request to B while our network adapter (green) is waiting for an ACK from a non existing node. The blue boxes indicate wait times according to the CSMA/CA protocol whereas the orange boxes are data send times. All times are in microseconds (denoted with t). The numbers in the bracket are the different steps in the CSMA protocol between A and B.

Software ACK timing In this paragraph, we discuss question number one. Experimentally, we determined a RTT of approximately 2.5 ms for our nodes. To have a safety margin to catch outliers, we let a stored sequence number expire after five ms (we consider the node unresponsive). This is a long time compared to the specification of CSMA/CA for IEEE 802.15.4 because the CSMA/CA protocol states that an ACK should be sent no later than $512\mu sec$ after sending the last bit of the data frame (see next paragraph). So, why do we have to wait that long and not just a bit longer than $512\mu sec$?

When mapping the ACK request and ACK has to be implemented in software, processing the ACK frame is delayed because an interrupt needs served and some code has to be executed before we can match the sequence number. This time can vary depending on the load on the network adapter. The other problem is that we already start time measurement before the data frame enters the CSMA protocol to get a useful measurement for the RTT. In CSMA, the measurement starts when the last bit left the radio chip.

In contrast: when implementing the ACK mapping in hardware, the timer is started when the last bit was sent and stops as soon as the frame is received. This allows to implement tighter timing constraints compared to software.

In summary, the answer to question one is: we wait for five ms until we consider the ACK overdue (sequence number expires). In the next paragraph, we discuss question number two.

Foreign ACK In this paragraph, we demonstrate that while waiting for the ACK from our RTT measurement, other nodes can exchange data and ACKs (*foreign ACKs*). Before we perform the calculations to demonstrate this, we require a formula which converts a number of symbols to the send time.

IEEE 802.15.4 at a radio frequency of 2450 MHz has a symbol rate of 62.5 ksymbols/s ([16], Subsection 6.5.3.2). Using this information, we can convert waiting times given in symbols to microseconds (Equation 4).

$$\frac{\#symbols\,[symbols]}{symbolRate\,[\frac{symbols}{s}]} \cdot 10^6 = time\,[\mu sec] \tag{4}$$

Using the symbol conversion formula and the frame definitions from the IEEE 802.15.4 specifications, we demonstrate next that our network adapter can observe a foreign ACK while it is waiting for a response to its own RTT measurement. We consider the following example: our network adapter executes an RTT measurement for a *non-existing* node N (i.e. sends an ACK request and waits for an ACK from N). In the meantime, a node A sends an ACK request to node B and B responds with the corresponding ACK. We calculate the time until our network adapter sees the ACK from B in the following way (Figure 14, the numbers correspond to the numbers in the figure):

1. Just after our network adapter sent the ACK request, node A has to wait a period called LIFS (Long Interface Spacing) before sending any data. The LIFS period is 40 symbols long which are $640\mu sec$.

2. A transmits a frame at 250 kbit/s. The size of the frame is 14 bytes (short addressing, no security headers, 1 byte payload, ACK request flag set). This operation takes $448\mu sec$.

3. Node B has at most $aTurnaroundTime + aUnitBackoffPeriod$ time to respond (32 symbols) to A's message, which corresponds to $512\mu sec$.

4. Node B sends the ACK at 250 kbit/s, which is 5 bytes long. This operation takes $160\mu sec$.

By summing up all these times, we get $1760\mu sec$. This is the time until our network adapter sees the foreign ACK in our scenario and it is well within the 5 ms timeout we set. In case the sequence number of the ACK from B matches the one our control unit used to try to reach N, our control unit could interpret the answer as an answer from N. This is what we call an *ACK collision*. In the next subsection, the issue of ACK collisions is discussed in the context of subnet device discovery.

4.2.6 Active device discovery

For device discovery, sniffed source and destination addresses serve as a source of information. However, we may not get the complete picture and hence we need active discovery as well.

The source and destination addresses of sniffed traffic serve as a source of information but do not provide a complete picture. Active device discovery is based on the same technique as measuring the RTT. We send frames with the ACK request flag set for all addresses in the subnetwork (*full sweep*) and check

```
1  r=0; //Global variable: current round
2
3  main()
4    for(r=0; r<255, r++)
5      for(s=0; s<255, s++)
6        set_sequence_number(s);
7        ack_request(0xrs); //Compose target address
8      pause(); //Wait for ACKs to arrive
9
10 //Callback function
11 on_ack_receive()
12   seq = get_ack_sequence_number();
13   print("found device: 0x" + r + seq);
```

Listing 5: Pseudo code for 802.15.4 device discovery using a full sweep. Part of the source address is encoded into the sequence number so the state we have to keep is reduced to one global variable.

whether we receive a response. As short addresses are 16-bit long, the range to scan consists of 65534 (0xfffe and 0xffff are reserved for special purposes). Although this brute-force approach is time consuming we are not aware of another procedure to discover 802.15.4 devices. Furthermore, we can only detect devices in range of the network adapter because we cannot join the network implemented by layers on top of 802.15.4.

We start by describing our first device discovery implementation where we tried to avoid storing the sequence numbers of ACK requests explicitly. Afterwards, we discuss the problem of ACK collisions and present a mathematical model. In the end, we describe our current implementation.

Avoiding state In our initial approach, we avoid to keep the mapping between an ACK request and the scanned source address by encoding part of the source address into the sequence number. For this, we split the 16-bit source address into two 8-bit blocks and encode the least significant 8-bit block of the address into the sequence number (Listing 5). This algorithm is resource and time efficient as we only need to store one variable.

The approach of sending 255 ACK requests and wait for an ACK suffers from one major drawback: we have no way to check whether the ACK is a response to our ACK request or if the ACK belongs to another data transmission. Hence, ACKs from foreign data packets will be misinterpreted as newly discovered device. This *ACK collision* is a result of the missing source address in an ACK packet and the limited number of sequence numbers.

Sniffing ACK collisions One could try to solve this problem by detecting foreign ACKs. By sniffing the sequence number of the original data frame, we could assign foreign ACKs to their corresponding data frame. However, this solution does not work when the sender of the data packet is out of radio range.

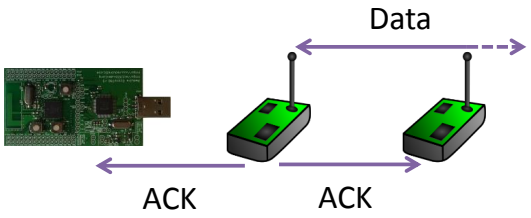

Figure 15: Foreign ACK with missing data context. If the sequence number of the ACK matches a pending ACK, it leads to a false positive because the ACK carries no sender address.

In this case, we miss the data packet but may still be able to see the related ACK (Figure 15).

Modeling ACK collisions The concept of ACK collisions has significant influence on the precision of our sweep. To better understand the relation between pending ACKs and foreign traffic, we present a mathematical model allowing us to analyze the expected number of ACK collisions. In our model, we assume that the traffic is uniformly distributed, that all foreign traffic is acknowledged and that the traffic follows the same probability distribution. This corresponds to the IID (Independent and Identically Distributed) assumption. If not all traffic is acknowledged, our model estimates the number of ACK collisions too high.

For our analysis we apply probability theory because every foreign ACK hits a sequence number of a pending ACK with a certain probability. This is a repeated yes/no experiment and hence we model it with a binomial distribution. When randomizing the sequence numbers, the repetitions of the experiment are independent. We define the following variables and functions:

- n: number of foreign ACKs intercepted during the sweep.
- pa: number of pending ACKs (e.g. if pa=20 we transmit 20 ACK requests and wait for a response until the ACK is overdue).
- sn: number of sequence numbers available (256 in case of 802.15.4).
- p: probability of a ACK collision, $p = \frac{pa}{sn}$.
- *mean(n)*: the mean number of ACK collisions.
- *std(n)*: standard deviations of mean(n).

Using the formulas for binomial distributions, we obtain the mean number of collisions (Equation 5) and the standard deviation (Equation 6).

$$
\begin{aligned}
mean(n) &= n \cdot p \\
&= n \cdot \frac{pa}{sn} \qquad (5) \\
std(n) &= \sqrt{n \cdot p \cdot (1-p)} \\
&= \sqrt{n \cdot \frac{pa}{sn} \cdot \frac{sn-pa}{sn}} \qquad (6)
\end{aligned}
$$

These two equations are of great value since they give us the chance to estimate the error rate of a full sweep. These equations do not contain implementation specific parameters and the relationship between the number of collisions and the amount of foreign traffic is linear.

One way to estimate the number of foreign ACK (not collisions) is provided by Equation 7, where d is the number of devices in our area, t the time for a full sweep (in minutes, implementation dependent) and m the number of messages on device sends per minute.

$$
n = d \cdot t \cdot m \qquad (7)
$$

To get an understanding about the order of magnitude, consider the following example: we have 30 sensors in range transmitting a measurement value every 15 seconds, our sweep takes 5 minutes and has $pa = 1$ (send request and wait for a possible response before continuing)

$$
\begin{aligned}
n &= 30 \cdot 4 \cdot 5 = 600 \\
m(600) &\approx 2.3\, collisions \qquad (8) \\
s(600) &\approx 1.5\, collisions
\end{aligned}
$$

In this scenario, we expect roughly two false positives per full sweep (Equation 8). Reporting two non-existing devices out of 30 sensors corresponds to an error of 7% which we cannot ignore. It gets even worse if we increase the number of pending ACKs to 20, then we get roughly 47 false positives with a standard deviation of seven.

Although this sounds discouraging at first, it is not so bad after all because we could reduce the number of possibly existing devices from 65534 to 32 (30 devices detected plus two false positives) for our first example. Having only a small number of devices left allows to apply more time consuming analysis methods. One could for example perform a round trip time measurement for each device marked as existing after the sweep. If a device replies, it is likely to be present because the probability for a collision is $4 \cdot 10^{-3}$ as the measurement duration is five milliseconds only.

Based on what we learned from the mathematical model, we describe our current implementation.

Current implementation Our current implementation for subnet device discovery is based on RTT measurements. Because the probability for an ACK collision increases with respect to the number of pending ACK, we decided to scan one device at a time (one pending ACK). This results in approximately two to three collisions per sweep in a "busy environment". We leave it up to the user to sort out the collisions. However, Smart Eagle assists the user by providing filter conditions and RTT measurement functionality.

4.2.7 Link quality

A measurement for the link quality is provided by the radio chip. The link quality indicator (LQI) is a value scaled between 0 and 255 indicating the signal strength. The exact implementation depends on the manufacturer of the radio chip and varies even between different chips of the same manufacturer. This is a valid behavior according to the IEEE 802.15.4 specifications, because they state ([16], Chapter 6.9.8): "The measurement may be implemented using receiver energy detection, a signal-to-noise ratio estimation, or a combination of these". We fetch this value as part of our sniffing functionality.

4.3 Interconnect

This subsection describes the protocol on the interconnect. We pursued two major design goals:

- Compact: all data is transmitted on the serial line and hence transmission speed is limited.
- Human readable: to allow for modular development and testing, a human readable format is desirable.

We decided to use the following base format: <prefix>=<csv> where <prefix> is the name of a parameter or command and <csv> are comma separated values. All numbers are in base 10 and time measurements in microseconds. A summary of the protocol is provided in the appendix (Subsection 9.2).

4.3.1 Contiki interconnect I/O

The Econotag USB connection is used as a serial console in Contiki. Receiving input from the serial console is event based: in the main function of our Smart Eagle process on Contiki, we wait for an input event to arrive, parse the command and the parameters and call the appropriate function afterwards.

Output is generated using printf, which offers convenient formatting options for variables. Although we utilize printf from different processes (Smart Eagle process and the system process handling incoming packets), there is no concurrency problem as the Contiki scheduler is non-preemptive [50]. Hence, a call to printf will never be interrupted by other processes.

4.3.2 Java interconnect I/O

When plugging in the Econotag into a PC running Ubuntu Linux (tested with Ubuntu version 11.10 and 12.10), it registers two serial consoles ("/dev/ttyUSBx", where "x" is a number). Our measurement protocol uses one of them. As devices are treated as files in Linux, we open the serial console as a file using a Java buffered reader.

Disconnects The first issue we encountered were periodic end of file messages from the serial console after a certain idle period. The blocking read in Java returned null and restarting the read operation was required.

After attaching the Econotag to the PC, the serial console in Linux is configured in *raw* mode which means that the data is forwarded without interpreting it. Reconfiguring the serial console to *cooked* mode using the Linux utility *stty* put things right. In cooked mode, control characters (like end of file) are interpreted by the system. The exact implementation of the cooked mode is however operating system dependent.

Losing characters While Java and the Smart Eagle application on Contiki were exchanging data, we encountered missing characters at irregular intervals. In most cases, it was only the first character of a message coming from a printf in Contiki.

At first, we suspected the serial line driver to be the problem. However, reimplementing the serial driver in a most basic version did not help. Increasing the buffer size for the Java buffered reader did not help either. Once we tried to fetch the input from a Linux shell script, the error did not appear anymore.

Unfortunately, we were unable to determine the cause of this error. Our workaround is to read the Contiki output from the deployment script as it displays the output from Contiki after successful deployment.

4.4 Control unit

This subsection describes the implementation of the IEEE 802.15.4 control unit. We start with an architecture overview and continue by explaining the building blocks of the control unit.

4.4.1 Architecture overview

The control unit relies on the measurement functionality implemented in the network adapter and its main purpose is to serve as a relay between the web interface the network adapter. We start with an overview about the major building blocks and discuss selected topics in more detail afterwards (Figure 16):

- Webserver: receive HTTP get requests and execute the corresponding measurement transaction.

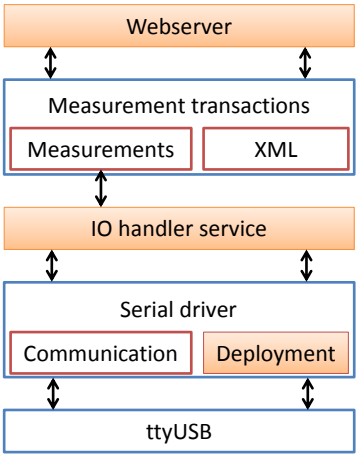

Figure 16: Major modules of IEEE 802.15.4 control unit. The beacon service and console handler are not shown. The modules marked orange are running in a separate thread. The red boxed indicate submodules used by the outer module.

- Measurement transactions: in charge of executing the measurement. All measurements are implemented in separate classes. The XML submodule offers the necessary data structures to hold the results and convert them to XML.
- IO handler service: executes measurements and receives sniffer data.
- Serial driver: in charge of Contiki deployment and offers an IO abstraction to higher layers.

4.4.2 Measurement transactions

Measurements are executed in a single thread on the network adapter because there is only one thread receiving and processing serial console input. Pending requests are automatically cached in the Contiki serial driver input buffer.

From the control unit down to the thread inside the network adapter, there is a lot of caching on different levels: web requests, Java buffered writer, OS, Contiki, etc. If the system fails under heavy load, the source is hard to determine and controlling the amount of parallelism inside the serial driver to prevent overwhelming Contiki is error prone. We decided to rely on the locking mechanism inside the transaction framework to prevent parallel requests.

4.4.3 IO handler service

The IO handler runs in its own thread and provides an interface to execute measurements and fetch sniffer data. Obtaining the sniffer data is simple: the

IO handler waits for sniffer data from the serial driver, decodes the message and stores the data into a data structure which can be converted to XML. When a transaction wants to execute a measurement or reconfigure the network adapter (e.g. the 802.15.4 channel), the following steps are executed:

1. The transaction thread creates on object M encapsulating the measurement object. This object is created from the corresponding class implementing the *ContikiCommandInterface*.
2. The thread executing the transaction invokes the *executeMeasurement* function on the IO handler to register the measurement and sends the actual command to the network adapter using the serial driver. This function call is blocking until one of the following cases apply (whichever happens first):
   (a) The measurement is completed successfully.
   (b) A timeout occurred.
3. After the function call returned successfully, M holds the measurement data in a form directly convertible to XML. If a timeout occurred, the measurement is marked as failed.

Data processing The IO handler thread processes all incoming data from the network adapter. For each incoming data line, the IO handler receive thread distinguishes several cases based on the prefix. The base idea is the following: if the data is sniffer data, it is directly processed and stored into a buffer to be fetched later. Otherwise the *processRequest* function on the registered measurement object is called. It can decide what to do with the data and if the measurement is completed.

4.4.4 Automatic deployment

To relief the user from manually deploying and starting Contiki on the network adapter, we implemented an automatic deployment mechanism as part of the control unit startup routine. A deployment script is called from Java which executes the same steps as when deploying it manually (Subsection 4.2.2).

Root permissions Unfortunately, we could not resolve the problem that deployment has to be executed as root user. We tried to reset the permissions for the serial console as well as adding the Linux user to the *dailout* group.

To enable automatic deployment anyway, the script needs to be executed as root without entering any passwords. This is achieved by setting the *setuid* bit enabling a script invoked by an unprivileged user to run with root privileges. However, for security reasons, Ubuntu ignores the setuid bit for shell scripts.

A typical work around for such cases is to invoke the script from a C program. Because the C program compiles to a binary, the setuid bit is not ignored anymore and the program runs with root permissions as desired.

5 KNX probe

In this section, we discuss the implementation details for the KNX probe. We start by describing the architecture and the network adapter. Afterwards, we discuss the interconnect and the implementation of the control unit.

5.1 Architecture

The KNX probe consists of a KNXnet/IP gateway attached to the KNX bus and a PC running the control unit software programmed in Java. The KNX probe offers link layer and application layer measurements as well as sniffing functionality. Our KNX network (part of the smart grid demo lab) uses TP 1 as a medium.

5.2 Network adapter

Compared to the IEEE 802.15.4 probe, we do not have to program the network adapter and the protocol on the intermediate network is already given (KNXnet/IP). This simplification comes with a major drawback: we have less control about the measurements we take. For the IEEE 802.15.4 network, the RTT measurement starts just before the frame enters the CSMA protocol and stops soon after the response frame arrives. In the case of KNX, the measurements cannot be executed on the KNXnet/IP gateway and the IP network in between cannot be eliminated.

At first, we were bothered by the presence of the IP network because it is an additional source of latency which needs to controlled. Yet, a KNX USB interface would not resolve the problem as USB suffers from the same problem as well [44]. Hence, whenever the network adapter cannot be programmed to perform the measurements, we need to consider the properties of the interconnect.

5.3 Interconnect

Because the interconnect cannot be ignored, having an IP network instead of a USB connection between the KNXnet/IP gateway and the control unit is an advantage. As discussed earlier (Subsection 2.4.1), measurements for IP based networks are a well-established topic in research and these results can be reused to control the influence of the IP network on the measurements.

This issue has not been investigated further because it is not our core focus and the solution consists of integrating already existing tools into our software (Subsection 8.1.2).

5.4 Control unit

In this subsection, we describe the implementation of the KNX control unit. First, we introduce the architecture. Afterwards, we discuss how we automatically locate KNXnet/IP gateways. Next, we present our implementation of

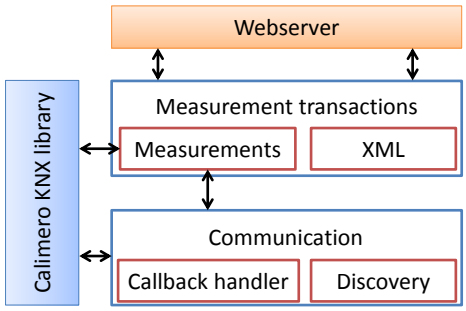

Figure 17: KNX control unit architecture. The red boxes indicate functionality used by the surrounding component and modules running in a separate thread are marked orange. The beacon service and console input handler are not shown.

the measurement functionality and sniffer. Subsequently, we briefly look into concurrency issues related to KNX measurements. In the end, we describe our KNX control unit simulator.

5.4.1 Architecture overview

The KNX control unit consists of four basic building blocks (Figure 17) which are briefly reviewed in a top-down manner before certain aspects are discussed in more detail.

Upon a HTTP get request, the webserver invokes a callback which executes the corresponding measurement transaction. Upon completion, the measurement results are packed into predetermined data structures (defined by us), converted to XML and sent back.

The communication layer is responsible for KNXnet/IP gateway discovery and adds a callback to Calimero for traffic sniffing. To communicate with the KNX bus and to implement the measurements, we use the Calimero library (Subsection 2.5.1). It maintains the connection with the KNXnet/IP gateway and offers a set of network, transport and application layer functionality to communicate with KNX devices.

5.4.2 Gateway discovery

To avoid that the user has to pass the IP address of the KNXnet/IP gateway to the control unit upon deployment, the gateways are located automatically. The control unit connects to the first KNXnet/IP gateway that comes along. The discovery procedure is part of the KNXnet/IP specification and implemented in Calimero. Upon completion, Calimero returns a set of discovered gateway IP addresses.

When establishing a connection to the Gateway, Calimero requires the IP address of the PC network interface attached to this network. As a PC can have

```
1  for each network interface i
2    for each KNX gateway address g
3      reachable = ping(g using i.getAddress());
4      if(reachable)
5        print(g reachable through i);
6    end
7  end
```

Listing 6: Pseudocode of the function to determine the source network interface given the KNXnet/IP gateways are known. It works by cycling through all possible combinations (brute force). The complexity of this algorithm is $O(i \cdot g)$ where i is the number of network interfaces and g the number of discovered KNXnet/IP gateways.

multiple active network interfaces, it is our job to figure out which one to use. Next, we discuss two different possibilities to find the IP address of the network interface through which the KNXnet/IP gateway is reachable.

Network prefix One possibility is to calculate the network prefix given the subnet mask and the target IP address. However, obtaining the subnet mask in Java given that only the destination host is known not straight forward.

The Java method to obtain the subnet mask is through an abstraction of a network interface. Yet, figuring out the right network interface for this particular IP address is the problem we are trying to solve. In order to obtain the right network interface, we would need the system routing table.

Ping Our solution utilizes the *ping* command which is available on most platforms and offers a second parameter to set the source IP. By cycling through each combination of host and gateway address, we obtain the local network interface (Listing 6). As a welcome side effect we automatically obtain a confirmation that the KNX gateway is actually reachable.

The drawback of this algorithm is its $O(i \cdot g)$ complexity, where i is the number of network interfaces and g the number of discovered KNXnet/IP gateways. However, the parameters i and g are expected to be around three which does not pose a problem. Furthermore, this algorithm could easily be parallelized.

5.4.3 Sniffing

Building a traffic sniffer with Calimero is convenient. After successfully establishing a connection to the KNXnet/IP gateway, we merely have to provide a callback implementing the *NetworkLinkListener* interface. It forces us to implement a function for *indication* and *confirmation* frames. Indication frames are received data frames and confirmation frames are link layer ACKs.

Our KNXnet/IP gateway supports only group monitor mode (not the bus monitor mode) and hence we can only monitor group communication. We can

for example not see link layer ACKs not intended for us.

5.4.4 Measurement functionality

Compared to the IEEE 802.15.4 probe, implementing the measurements is easier because the basic functionality is provided by Calimero. The challenge is to understand the exact behavior of KNX on the link layer, network layer and partially the application layer.

Link layer RTT To provoke a link layer ACK from the KNX bus, we manually assemble a KNX frame containing a one byte payload with content "0". Afterwards, the *sendRequestWait* function from the Calimero library is executed to send the frame and wait for a link layer ACK or timeout (blocking function call). In the end, the ACK frame is fetched from the sniffer (confirmation frame in Calimero callback handler).

Although the sendRequestWait function returns when the frame is ACK or throws an exception when there is no ACK, we still need to fetch the ACK frame from the sniffer as a link layer ACK can either be positive or negative. We want to distinguish both cases and report the results to the measurement application.

Connectionless RTT A connectionless RTT measurement is based on a device descriptor read operation specified in the KNX application layer ([10], Section 3/3/7). This is the closest we get towards a network layer RTT measurement. The device descriptor read operation returns an four byte value (ignored by the control unit). If the remote device does not exist, the request times out after one second which is the lowest value that can be configured in Calimero.

Connection-oriented RTT Connection-oriented communication is the official procedure for KNX device discovery ([10], Section 3/5/2). It attempts to establish a connection to the remote device. If the remote device exists, it responds with a disconnect and if the device does not exists, no response is received.

We slightly modified this procedure and perform a device descriptor read operation after the connection has been established. This gives us the following two advantages:

- It is ensured that the connection has been completely established and that the device is capable of interaction using this connection.
- It makes our measurement comparable to the connectionless case as the request and payload is the same.

However, this is even further away from a network layer RTT measurement because multiple messages are exchanged.

5.4.5 Concurrency

Although our measurements are based on the Calimero library which is thread-safe, there are two reasons which force us to implement our own concurrency control mechanisms through our transaction framework.

Calimero bug During our preliminary tests, we discovered a bug in Calimero leading to wrong results in case of parallel connectionless RTT measurements. Devices which do normally not respond upon a connectionless device descriptor read operation suddenly respond if a connectionless request to another device is running in parallel. We developed a small, self-containing test case to demonstrate this behavior and submitted a bug report. The connection-oriented mode is unaffected.

We avoid the problem by the aid of the transaction framework. A global lock prevents that two connectionless requests are running in parallel.

Parallel link layer RTT Another problem are parallel link layer RTT measurements with the same target address because data and ACK frames do not carry a sequence number. If we send two measurement frames A and B, one after each other, to the same target address, it may happen that the ACK for B (B.ACK) arrives before A.ACK at the sniffer callback. By mistake, B.ACK would be assigned to request frame A and vice versa.

To prevent concurrency in this particular case, a key-based lock is implemented (fine-grained locking), i.e. each target address is locked separately allowing parallel requests to different target addresses.

5.4.6 Simulator

For development and testing purpose, we implemented a simple simulator into the KNX control unit allowing it to run independently from a KNX network.

Measurement requests return random values but always succeed. For the sniffer, we recorded the traffic on the KNX bus for a few hours and wrote the resulting XML data into a file. The simulator replays this data in an infinite loop.

6 Measurement application

This chapter describes the software architecture and implementation of the Smart Eagle measurement application. First, an architectural overview is provided and afterwards the three major building blocks are presented.

6.1 Architecture overview

To prevent mixing GUI code with the program logic, Smart Eagle uses the popular model-view-controller pattern (Figure 18). In our case, the model is represented by the data storage holding all the measurement data we gathered. The measurement functionality is the controller part and responsible for the interaction between the control units and the application.

The graphical user interface displays the measurement data according to the state of the data storage and the data storage gets directly updated by the corresponding measurements.

6.2 Data storage

Every network type has a root node and a set of associated children in the data structure (Figure 19). The root node inherits a set of common functions from the *RootNodeBaseFunctionality* class concerning network and control unit information. We implemented a different data structure class for each type of network to accommodate for the varying network structures and to avoid frequent type casting. A separate instance of the corresponding data structure exists for each registered control unit.

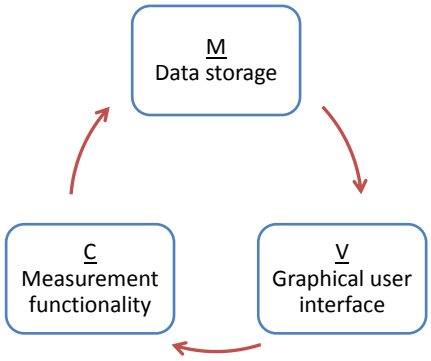

Figure 18: MVC pattern in Smart Eagle. The measurement functionality acts as controller and the data storage as model.

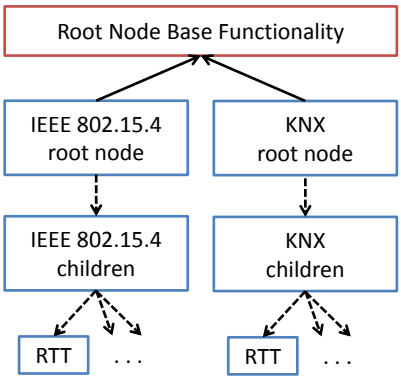

Figure 19: Architecture of the data structure storing the measurement results. The red box indicates an abstract class containing common functionality for the root nodes. The solid arrows represent an inheritance relationship whereas the dashed arrows are references to other objects.

6.2.1 Root node

In our setting, the root nodes do not have a corresponding node in the network - they are an artificial construct to group everything together which belongs to that particular network. They store different information regarding the network as a whole and the associated control unit, for example:

- A human readable network name (displayed by the GUI).
- Various information about the control unit, such as its network address (intermediate network), connection status, time of last contact.
- A ring buffer containing the logs received from the control unit.

The implementation of the root node is very flexible because every network implements its own data structure. Next, we quickly review the peculiarities of the root nodes for IEEE 802.15.4 and KNX networks.

IEEE 802.15.4 To store the children, the root node holds a concurrent hash map. As a key, an IEEE 802.15.4 address is used and the value is a reference to a child object containing the measurements. Because the network adapter can be configured (address, channel and panID), the root node contains the current network adapter configuration.

KNX An N-ary tree would be the ideal data structure to represent a KNX network. However, the Java library does not offer an N-ary tree. To keep our implementation simple, we decided to use a concurrent hash map to store the children and provide a tree based interface to the users of the data structure.

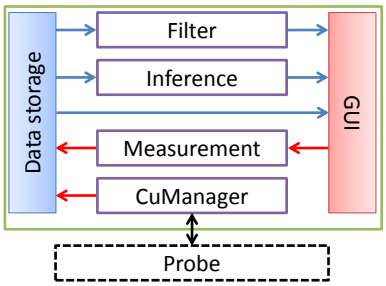

Figure 20: Data flow from data storage to GUI (blue) and vice versa (red). The components in the middle between data storage and GUI belong to the measurement functionality package of the corresponding network. The components belonging to the measurement application are surrounded with a green box.

6.2.2 Child node

The children are represented by *DeviceNode* objects and each object corresponds to one node in the NUT. The only purpose of a device node is to keep the measurement information. It does deliberately not offer any methods to retrieve interpretations about the measurements because we want data storage to be separated from data analysis. This allows to outsource the data storage to a database for increased capacity and permanent storage. Methods for data processing are offered in the measurement functionality package as shown in the next subsection.

6.3 Measurement functionality

The measurement functionality package acts as a controller in the MVC pattern and is implemented separately for each network type. Its purpose is to gather data from the control units (logs, sniffer data), obtain measurements, offering interpretations of the measurements (inference) and filter functionality (Figure 20). The next subsections discuss each module in more detail.

6.3.1 CuManager

For each connected control unit (advertised through a beacon), a control unit manager (CuManager) running in a separate thread is launched. It performs periodic tasks like fetching sniffer and log information. This data is written directly to the data storage.

If the connection between the application and the control unit is interrupted, the CuManager sets a disconnected flag, notifies the GUI and shuts down. Upon reconnection, a new CuManager is launched and the old data structure is reused meaning that the data is still there.

6.3.2 Measurement

The measurement functionality is implemented on top of the measurement transactions offered by the control units. In the next paragraphs, we discuss the cases where we have to do more than just sending one measurement request to the control unit.

KNX device discovery The KNX specifications define that a sweep has to be executed by contacting to each node in the subnetwork in connection-oriented mode ([10], Section 3/5/2). Hence, we cycle through each address as well but in contrast to the specifications we offer multiple different options: connection-oriented mode, connectionless mode or link layer.

An option to skip known devices has been implemented as well. A device is marked as known if it sent traffic before or an application layer RTT measurement has been performed. We do not consider a device existing if only a link layer ACK has been received due to the semantics of TP 1 bridges and repeaters (Subsection 2.1.5). This allows to reuse information from different sources like the sniffer to speed-up the discovery process. To further accelerate the process, the number of threads executing RTT measurements in parallel can be configured.

KNX topology discovery Topology discover is similar to device discovery, except that it only performs RTT measurements for area and line couplers instead of all devices. To speed-up the process, a *smart scan* option allows to skip a subtree if its parent does not exists.

IEEE 802.15.4 sweep The sweep for an entire subnet is already implemented in the control units. Hence, this process is simply repeated for all 256 subnets.

IEEE 802.15.4 channel scan The channel scan listens for sniffer data on each channel during a configurable time and counts the number of frames received. When the channel scan mode is activated, no data is written to the data storage to avoid cluttering it up with data from different channels.

Monitoring The base idea of monitoring is to frequently check if the device still exists using an active measurement approach. If monitoring is activated, it is applied to all currently known devices, but not to newly detected devices during monitoring. The following three parameters are configurable:

- Timeout: wait time between two successive scan operations. The higher the timeout, the lower the network load but the longer it takes until an unresponsive device is detected.
- Missing: number of missed RTT measurements before a device is considered offline. The higher the value, the fewer false positives but the higher the chance that we miss short outages of a device.

- Rescan: the algorithm periodically rescans devices marked as offline. If a device responds, it is marked as online again and monitored. Cycling through all online devices is defined as one round. With the rescan value, the user can configure after how many rounds the list of offline devices should be rescanned. The lower this value, the less time to monitor online nodes but the faster we detect offline devices which are online again.

For KNX, we only implemented the *timeout* setting. However, the other two options can be implemented analogous to the IEEE 802.15.4 case.

6.3.3 Filter & inference

The graphical user interface can either fetch the measurements directly or through an inference function (Figure 20). The inference function operates on the data storage to combine and interpret the results. A typical example is a function which calculates the total traffic sent by a certain device in bytes or counts the number of known devices for a certain network.

The filter functions are applied directly to the JTree to display only nodes satisfying certain properties. Smart Eagle implements filters for the following properties: incoming traffic, outgoing traffic, link layer response and application layer response. They can be chained together and the whole filter can be negated as well. An example of a filter in Smart Eagle to find devices which are only discovered through sniffing is shown in Equation 9.

$$\begin{aligned} f(d) &= \neg(LL(d) \vee NL(d)) \; where \\ d &= device \\ LL(d) &= True \; if \; d \; contains \; positive \; link \; layer \; response \\ NL(d) &= True \; if \; d \; contains \; positive \; network \; layer \; response \end{aligned} \tag{9}$$

6.4 Graphical user interface

The GUI is split into four different viewing areas (Figure 21):

- JTree: display known nodes according to data storage and applied filters.
- Logs / progress: The *application logs* tab consecutively shows the application logs. The *Measurements* tab indicates the status of currently ongoing measurements by displaying a progress bar. A measurement can be canceled by pressing the cancel button located next to the progress bar.
- Info area: present the measurement information associated with the selected node. If the root node is selected, information about the network is shown.
- Button panel: depending on which node is selected in the JTree, all applicable measurement operations and their configuration options are presented. Similar measurement functionality and their configuration are clearly laid out by grouping them together.

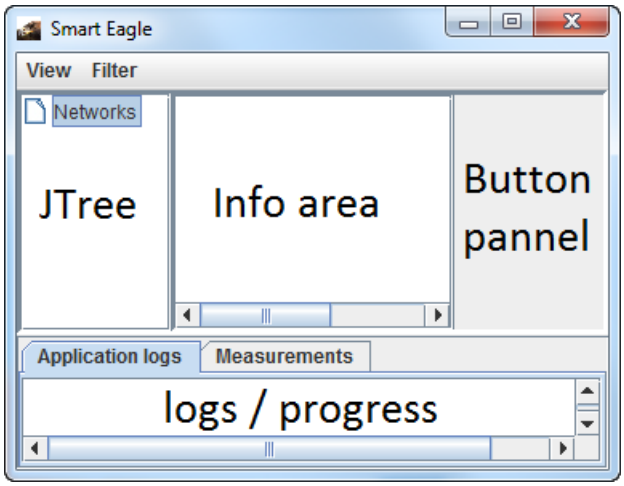

Figure 21: GUI skeleton: the main window is split into four components. The JTree displays the network nodes, the info area the related network measurements and the button panel offers network measurement functionality for the selected node. The bottom panel shows application logs and to progress of pending measurements.

The JTree, Logs tab and Measurements tab are considered static because they are not rebuilt when the user selects a different node in the JTree. However, the content can change as a response to other actions like starting a measurement. The content of the info area and button panel depend on which node is selected and is generated from scratch each time the user selects another node in the JTree.

Tool tips We decided to implement the help text as tool tip instead of an extensive user manual. On our opinion, this is more convenient because the user gets help by hovering over the element on which he is focused. However, this does not replace a user manual providing an overview about the functionality of Smart Eagle. In our case, the current document explains the Smart Eagle functionality in detail and Subsection 3.3 provides a functional overview.

6.4.1 Implementation

The software architecture of the GUI reflects the previously shown structure. In particular, the static components are separated from the context aware components and the context aware components are subdivided by network type (Figure 22).

Context awareness The context aware components register for JTree selection events. Whenever the user selects a node, the network type is determined

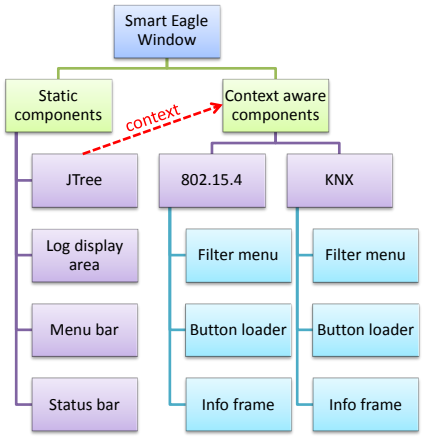

Figure 22: GUI software architecture: we distinguish between static and context aware components (green). The context aware components are displayed depending on which network is selected in the JTree (red, dotted arrow) and there is an implementation for each network type (blue).

and the content is generated accordingly. Although the principle is simple, the implementation is tricky because one cannot use a graphical GUI builder to generate and place the components. Hence, we programmed the GUI mostly by hand. Next, we explain our GUI programming approach using the example of the ButtonLoader.

ButtonLoader The ButtonLoader is a JFrame located on the right side in our GUI (Figure 21). Our class, *MeasurementButtonManager*, extends the JFrame class and adds various methods for adding spaces, boxes containing a set of components, etc. To provide insight into our framework, Listing 7 shows a minimal GUI programming example (Java pseudocode) using the Measurement-ButtonManager. The four major steps are:

1. Line 1: create an empty list to hold all elements belonging to one group in the ButtonLoader panel.

2. Lines 3 to 6: setup a label and text field next to each other (the label provides the description for the text field). These two components are aligned horizontally and added to the GUI element list.

3. Lines 8 to 10: create a button to launch a measurement and add it to the GUI element list.

4. Line 12: setup a bounded group with a name and the elements stored in the GUI element list. The framework creates a group of elements surrounded with a line and adds a title (Figure 23).

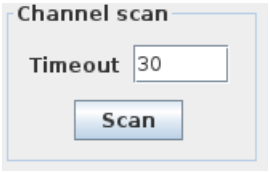

Figure 23: The GUI component generated from the code shown in the programming example (Listing 7).

Expanded tree We encountered a common JTree problem while adding new nodes to the tree: the tree collapses. The reason is that adding new nodes invalidates the tree structure and it has to be redrawn. A routine solution is to store the tree state, add the node and then apply the tree state again [51].

6.4.2 Graph view

The graph view has been implemented using the JGraph[10] library for Java. We use a hierarchical layout to display the KNX network structure. Initially, we wanted to show the IEEE 802.15.4 network in the same graph as KNX. However, we were unable to configure JGraph in such a way that both graphs are displayed next to each other such that KNX is rendered hierarchically and 802.15.4 as a circle. Because graph rendering is out of scope, we only show the KNX network in the graph.

[10] http://www.jgraph.com/

```
1  LinkedList guiElements = new LinkedList();
2
3  JLabel timeout = new JLabel("Timeout");
4  JTextField timeoutVal = new JTextField("30");
5  guiElements.add(
6          MeasurementButtonManager.packAndLayout(timeout, timeoutVal));
7
8  JButton button = new JButton ("Scan");
9  button.setActionListener(...);
10 guiElements.add(button);
11
12 buttonPannel.addComponentGroup("Channel scan", elementList);
```

Listing 7: GUI programming example illustrating how to use the MeasurementButtonManager framework. The example has a label, text field and button.

7 Evaluation

Throughout our evaluations, we used the following equipment to run Smart Eagle:

- Lenovo-PC: dual core Intel Pentium G6950 (2.80 GHz) with 3 GB of RAM. It is equipped with a dual boot:
  - Ubuntu Linux 11.10 i386 with the Java openjdk-i386 1.7.0 (used to evaluate Smart Eagle)
  - Windows XP with ETS 3.0f
- FitPC: dual core Intel Atom Z530 (1.60 GHz) with 1 GB of RAM. It runs Ubuntu Linux 12.10 i676 with the Java openjdk-i386 1.7.0_09.

For networking, we used a 10 Mbit Ethernet hub from Netgear (EN 104 TP) connected to the ABB smart grid demo lab and interconnecting both machines.

7.1 Application & system

In this subsection, we evaluate Smart Eagle in terms of stability and review our design decisions.

7.1.1 Stability

The testbed consisted of an IEEE 802.15.4 control unit and a KNX control unit attached to the smart grid demo lab. The measurement application and the KNX control unit were both deployed on the Lenovo-PC. The IEEE 802.15.4 control unit was installed on the FitPC. To generate 802.15.4 traffic, we used two Plugwise and the Plugwise USB dongle (plugged into the FitPC).

We ran the system for approximately 50 hours without restarting any component. Smart Eagle was permanently sniffing traffic. We also generated heavy load for about one hour by running multiple network monitoring instances in parallel.

We found that the control units and the application were stable and responsive. However, we detected an odd addressing behavior from the Plugwise which is described in more detail in the next paragraph.

Plugwise network behavior We discovered that even though we only had three IEEE 802.15.4 devices in our test network, Smart Eagle recorded 2136 different addresses, most of them were short addresses. Using device discovery, we found that in the end only four addresses were actually reachable: two short addresses, one long address and the broadcast address ff.ff. The number of reachable addresses corresponds to the number of Plugwise in our network (except for the broadcast address). Next, we present a short analysis of the broadcast address behavior and the Plugwise address assignment.

The broadcast address is reachable through RTT measurements and we only get one response upon an RTT measurement. By using the link quality value and moving around the Plugwise, we discovered that only one device responds to the RTT measurement on the broadcast address. When unplugging this device, another device responds on the broadcast address. When all devices are unplugged, we get no RTT response on the broadcast address. This behavior is surprising as according to the IEEE 802.15.4 specifications, frames sent to the broadcast address should not be ACK by any device ([16], Subsection 7.5.6.2).

Regarding the addressing behavior of the Plugwise, we suspect that one of the Plugwise is not configured. We could reproduce the addressing behavior when using two Plugwise and the Plugwise dongle. We frequently detect new short addresses having only three outgoing frames. However, an in-depth analysis of this behavior is out of scope.

7.1.2 Transactions

While working with the Smart Eagle user interface, we felt that waiting for a long running measurement to complete (e.g. 40 min) without seeing partial results is unsatisfactory - the user gets inpatient. It is for example better to see newly discovered nodes added to the JTree on the fly. Our initial idea of always presenting a measurement as a single transaction to the user is not the right way.

Our current approach for communication with the user through the GUI is to show partial results (e.g. update the JTree when we detected a node while performing device discovery). In case of an error while performing the measurement, we show a dialog box to make the user aware of this.

We kept the idea of treating the communication between the measurement application and the control unit as a transaction. This is comfortable for the programmer as this transaction based model enforces a clear semantics.

7.1.3 GUI

The JTree on the left side of the Smart Eagle GUI is clearly arranged and shows the discovered nodes. The graph view is not very useful in its current form due to the width of the graph and there are better ways to visualize the network (Subsection 8.1.2).

The context aware elements, especially the button panel on the right, are a good choice. Presenting only the measurements which are actually available for the selected node provides the user with a clear and concise view (Figure 24).

7.2 IEEE 802.15.4

This subsection evaluates the measurement functionality of the IEEE 802.15.4 probe. The control unit is running on the FitPC and the measurement application runs on the Lenovo-PC. We placed two Plugwise and the Plugwise stick in the radius of 50 cm around the network adapter.

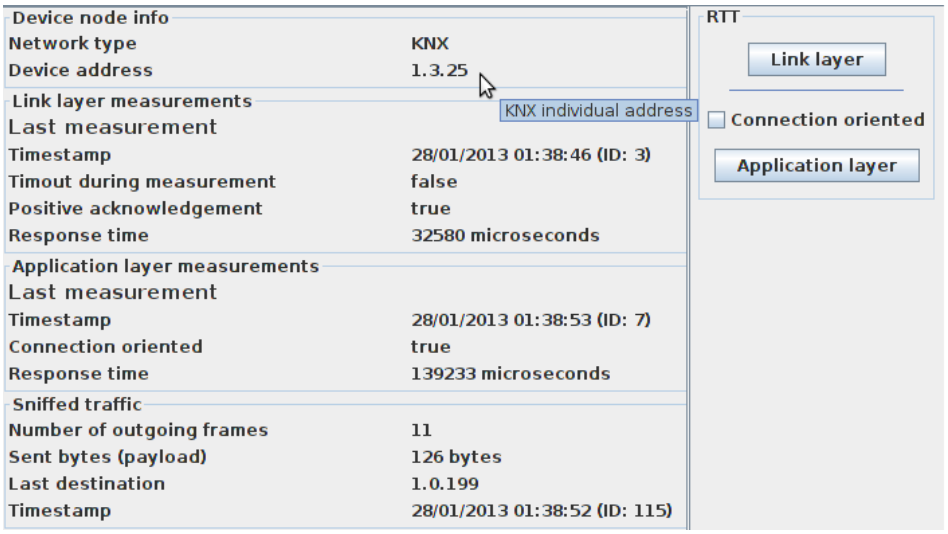

Figure 24: Smart Eagle screenshot, a KNX node is selected in the JTree. The subnet scan option is not visible because a device node is selected (not a network element). The information frame shows details about the last measurement and the sniffed traffic. Below the mouse pointer, a tool tip text is visible.

7.2.1 Device discovery

The Smart Eagle device discovery feature for 802.15.4 networks discovered all three devices plus the broadcast address ff.ff. We repeated the discovery process three times and it took on average 4 *min.* 30 *sec.* (standard deviation: 1 *sec.*).

Using the sniffer, we found that the Plugwise devices are quite active. When sniffing for 1 *min.* we detected all three devices plus the broadcast address.

Sweep command speedup As we implemented a separate sweep command on the network adapter, we want to know whether this was worthwhile. The alternative is that the measurement application performs a RTT measurement for each individual address separately. Using this approach, the discovery time increased significantly to 39 min. and 50 sec. with a standard deviation of 10 sec.

This demonstrates the overhead associated with each measurement request as the actual RTT measurement on the network adapter takes 2 - 5 ms. However, invoking an RTT measurement from the measurement application takes around 50 - 60 msec. The additional delay comes from the intermediate network but mostly from the processing overhead on the measurement application and control unit (webserver, transaction, etc.).

7.2.2 Monitoring

The two main criteria for a monitoring service are its detection rate (i.e. did it detect all outages?) and the time until such an outage is detected. To test these properties, we unplugged devices while the monitoring service was active. To reduce the measurement error, we started the time measurement using a button in Smart Eagle as soon as we plugged out the device. The time when the failure is detected was recorded automatically.

During our test, Smart Eagle discovered all outages and the detection times are summarized in Table 5. We modified the following two parameters:

- Timeout: the time the measurement application waits between two successive RTT measurements for monitoring purpose. The higher this value, the less load on the network but the longer it takes until an outage is detected. This is a linear correlation because doubling the timeout means that it takes twice as long until an address gets rescanned.
- Missing: the number of RTT measurements before a device is considered missing. Increasing this value reduces the number of false positives. Again, the correlation between the time until an outage is detected and missing ACK is linear.

7.2.3 Channel scan

The channel scan functionality has been tested with one minute wait time per channel. We expected traffic on channel 4 because it is used by the Plugwise. However, the test results showed traffic on channel 15 as well. We investigated this in the following way using Smart Eagle (providing an example of how to use Smart Eagle to analyze unknown networks):

1. Switched the network adapter to channel 15.
2. Observed the traffic using the sniffer information.
3. Switched to the correct panID obtained through the sniffed traffic.
4. Set a long address because the nodes obtained through sniffing all had long addresses.

Missing	200 ms	400 ms
4	3.4 ± 0.3	6.3 ± 0.3
2	1.3 ± 0.3	2.5 ± 0.3
1	0.6 ± 0.3	1.1 ± 0.4

Table 5: IEEE 802.15.4 monitoring evaluation. We measured the time until a failure was detected. For each row, we changed the number of "missed RTT measurements". Each measurement has been repeated 5 times and the time entries have the following format: "seconds.milliseconds".

5. Performed for each node a few RTT measurements to see if they respond.

In contrast to the Plugwise next to the network adapter, the newly detected nodes had a much lower link quality value (LQI) of 40 compared to 150. The maximum LQI value returned by the radio chip is 255. This lead to the conclusion that the unknown nodes had to be further away. In fact, some Econotags were located in the office next door running an experiment involving IEEE 802.15.4 communication and we picked up their traffic as well.

7.3 KNX

Again, the control unit runs on the FitPC attached to the smart grid demo lab by Ethernet. There are at least two Ethernet switches between the KNXnet/IP gateway and the control unit, one of them is our Netgear switch. The measurement application runs on the Lenovo-PC.

7.3.1 Topology discovery

A topology discovery feature is not available in ETS so we compared our results with the project database containing our KNX network configuration. Smart Eagle offers two types of topology discovery:

- Full scan: scan all addresses possibly assigned to network devices.
- Smart scan: scan only subnetworks having a parent.

Both scans discovered one area coupler having a line containing four line couplers. As expected, the smart scan (5:08 min $\pm$ 0 sec) was much faster than the full scan (50:40 min $\pm$ 1 sec). We repeated each measurement three times.

7.3.2 Discovery accuracy

To see how reliable Smart Eagle detects KNX devices, we evaluated the subnet device discovery feature on each of the four subnetworks. Device discovery was

Subnet	ETS DB	SE miss	SE extra	Miss conf.	Extra conf.	Total
1.0.0	11	1	0	1	-	10
1.1.0	59	0	0	-	-	59
1.2.0	43	0	3	-	3	46
1.3.0	43	1	0	1	-	42
1.4.0	50	0	0	-	-	50

Table 6: Smart Eagle discovery accuracy compared to the ETS project database (ignoring a dummy device with address 1.0.104). The numbers in the table represent the number of devices. Legend: database (DB), Smart Eagle (SE), missing nodes (miss), additional nodes (extra), confirmed by ETS device scan (conf), not applicable (-).

performed in connection-oriented mode with one thread only to maximize the discovery precision. We repeated each measurement three times.

Baseline We compared the results of Smart Eagle against the ETS project database and found several discrepancies (Table 6). To figure out if Smart Eagle or the KNX project database is wrong, we applied a device scan to each device in question using ETS. The ETS device scan checks if a single device address exists. In all cases, Smart Eagle was correct and we found five errors in the project database of the ABB smart grid demo lab. We are surprised that five out of 200 device entries in the ETS database are wrong because all device deployment and configuration is done through ETS.

ETS subnet scan ETS offers a device subnet scan and we applied it to each subnetwork as well (Table 7). The average discovery rate was between 40% and 60% which a high standard deviation. We conclude that in case of device discovery, Smart Eagle performs much better than our version of ETS.

Problem analysis We contacted KNX specialists at *ABB Stotz-Kontakt GmbH* in Germany to discuss the issue and they analyzed the scenario in their lab. We figured out that the bus load is the problem: our line and area couplers are configured to repeat telegrams up to three times if there is no link layer ACK from the target. This means that each device is contacted four times if it does not exists, generating a high bus load. It seems that this problem has been resolved in newer versions of ETS. However, as we received the updates just before the end date of this thesis, there was no time for a in depth analysis.

7.3.3 Performance & threading

Next, we compared ETS with Smart Eagle regarding discovery speed and accuracy (Table 8). For Smart Eagle, we used a different number of threads (1 - 4) to observe the influence of threading. The two results of this experiment are:

- Doubling the number of threads cuts the discovery speed by half.
- When increasing the number of threads, discovery accuracy decreases.

We may face the same issue as discussed in conjunction with ETS (Subsection 7.3.2). However, it is surprising that Smart Eagle already starts missing devices when only having two parallel measurements. This would mean that the bus is already overloaded when executing two parallel connection-oriented requests.

7.3.4 Connectionless discovery

As mentioned in Subsection 2.1.4, not all devices support connectionless communication. To determine how many devices are affected, we ran the discovery on subnet 1.1.0 and 1.2.0 (Table 9). Although discovery is faster than in connection-oriented mode due to lower timeout values, this method is not usable for device discovery as many devices are not detected.

7.3.5 Data source combination

Discovering the topology using the sniffer only worked well as we picked-up traffic from each subnetwork (Table 10). The number of detected devices varies between different subnetworks depending on the activity of the devices. Furthermore, it is not surprising that the number of detected devices increases when sniffing for a longer period. The standard deviation is lower when sniffing longer because the number of events (e.g. pressing a light switch) are more balanced over time.

If we run a device discovery with two threads on the 1.1.0 subnet after 15 min of sniffing and skip the known devices, discovery time is only reduced by $3\ sec. \pm 1.1\ sec.$

7.3.6 Link layer analysis

As expected due to the KNX link layer ACK semantics (Subsection 2.1.5), a link layer sweep is not usable for device discovery. When running a topology discovery, all network elements except the ones in the 1.0.0 line are marked as existing. In the 1.0.0 line, only the actually existing network elements (line couplers) are found. The difference between the 1.0.0 line and the other lines is that our KNXnet/IP gateway is member of the 1.0.0 line. Hence, the probing traffic does not pass the 1.0.0 area coupler for device discovery inside the 1.0.0 line.

We checked the 1.0.0 area coupler device configuration and found that it is configured to ACK all frames (filtering is disabled). Furthermore, we found that filtering in the line couplers of the 1.0.0 line is disabled as well. This corresponds to our measurements, showing all addresses in the 1.1.0 to 1.4.0 subnet as existing.

We conclude that link layer sweeps are not suited for device discovery but depending on their position in the KNX bus, we can learn something about the configuration of the network elements. Link layer analysis could be used as building block to find non-optimal KNX network configurations (e.g. filtering disabled, errors in the filter table).

Subnet	Mean #discovered devices	stddev (devices)
1.1.0	36 (61 %)	8
1.2.0	25 (58 %)	11
1.3.0	18 (42 %)	1
1.4.0	20 (39 %)	8

Table 7: Discovery accuracy of ETS for each KNX subnet of the ABB smart grid demo lab. We repeated each measurement four times.

	ETS	SE (1)	SE (2)	SE (3)	SE (4)
Time (min:s)	0:32	39:59	20:07	13:29	10:14
Time variance (s)	0:2	0:1	0:3	0:3	0:1
Mean #discovered devices	36.0 (61 %)	59.0 (100 %)	58.75 (99.6 %)	58.3 (98.2 %)	55.25 (93.6 %)
stddev (devices)	8.0	0.0	0.5	1	1.3

Table 8: Performance and accuracy comparison between Smart Eagle (SE) and ETS for the 1.1.0 subnet. The number of threads is indicated in brackets. We repeated each measurement four times. The time entries have the following format: "minutes:seconds".

Subnet	Mean #discovered devices	stddev (#devices)	Time (min:s)	stddev (s)
1.1.0	1 (2 %)	0	4:29	0
1.2.0	20 (47 %)	0	4:12	0.6

Table 9: Accuracy of KNX subnet device discovery in connectionless mode. We repeated the measurement for each subnetwork three times.

Subnet	Sniffing (5')	Sniffing (15')
1.0.0	9 (90 %), $stddev:\ 0$	9 (90 %), $stddev:\ 0$
1.1.0	9 (15 %), $stddev:\ 6$	15 (25 %), $stddev:\ 1$
1.2.0	17 (38 %), $stddev:\ 3$	21 (46 %), $stddev:\ 3$
1.3.0	16 (39 %), $stddev:\ 3$	23 (56 %), $stddev:\ 2$
1.4.0	6 (11 %), $stddev:\ 4$	12 (23 %), $stddev:\ 2$

Table 10: Accuracy of KNX device discovery by sniffing only. In the first run, we sniffed for 5 min and in the second run for 15 min. The numbers in the table represent the number of discovered devices. We repeated each measurement three times.

7.4 Known issues

We are aware of the following issues in Smart Eagle:

- Contiki
  - The Contiki timer precision is limited to 80 microseconds. An idea for improvement is presented in Subsection 8.1.1.
  - The LQI value for sniffed traffic reported by Contiki is incorrect. We believe that this is a bug in Contiki, in particular because ACKs, as a response to RTT measurements, provide the correct LQI value. Hence we can still evaluate the link quality by using RTT measurements.
- Measurement application
  - The measurement application does not delete any data from the data storage and eventually it will run out of memory. However, it runs stable for at least 50 hours (Subsection 7.1.1). This issue could be solved by storing old data in a compressed format, e.g. instead of keeping each sniffed frame we received, we only keep one counter to store the number of received frames.
  - The currently selected node in the JTree does not stay selected when a node is added to the same hierarchical level or a parent. If the selection in the JTree is lost while configuring measurement options, the new configuration is not applied to the measurement. Selecting the node and configuring the measurement again resolves the problem.
- Control units
  - There is no support for multiple measurement applications attached to one control unit. This leads to a race condition when fetching logs and sniffer data (the first one who fetches the data gets it). However, measurement operations are not affected.
  - Currently, we support only one control unit per IP address as the webserver runs on a fixed port. This could be resolved by dynamically selecting a free webserver port and including it into the beacon.
  - The webserver binding does not work correctly when multiple network interfaces are active on the control unit.

8 Conclusions

In this thesis, we explored the design of a measurement system for heterogeneous networks in smart grid environments. We developed a prototype of such a measurement system to gather experience and evaluated our approach.

During our study of existing systems and the smart grid requirements, we worked out a system architecture based on desirable and necessary properties with respect to smart grid network measurements. We propose to split the system in three parts: measurement application, control unit and network adapter. Furthermore, the networks relevant for measurement are the intermediate network and the network under test.

By implementing a control unit for KNX and IEEE 802.15.4, we demonstrated that our external, distributed network measurement approach is feasible. This has been confirmed by comparing our system against other products specifically designed for one network type.

For IEEE 802.15.4 networks, we presented a mathematical model to analyze the impact of ACK collisions while searching for devices in range. During the analysis of our KNX probe, we discovered that: the ETS project database is outdated, the subnet sweep of our ETS version has a flaw and that filtering for certain network elements is disabled.

Our last contribution is the design of the measurement application including a GUI. We determined that such a GUI has context-aware components as well as static components. Furthermore, we found that a traditional hierarchical graph leaks usability for the end user due to its width.

8.1 Future work

Smart Eagle in its current form is stable and can analyze KNX and IEEE 802.15.4 based networks. It furthermore presents a simple but clearly arranged GUI helping the user to gather and interpret network measurements. However, Smart Eagle is not a mature piece of software ready for deployment and we can think of many improvements and extensions which are surveyed in the following subsections.

8.1.1 Improvements

In this subsection, we discuss how to improve the existing software and the currently available measurement functionality.

KNX interconnect The interconnect link between KNXnet/IP gateway and control unit is currently ignored. This Ethernet based network including switches is a source of delay and delay variation (jitter) which needs to be considered to obtain more precise measurements.

In a first step, influencing factors like switches should be removed by attaching the control unit directly to the KNXnet/IP gateway using an Ethernet crossover

cable. Next, common utilities (e.g. ping or iperf [11]) can be used to characterize the interconnect with respect to RTT and jitter. As KNX is orders of magnitude slower than the Ethernet connection and there are no network elements on the interconnect, we expect a low jitter. Hence, subtracting the average latency should provide good results.

Another approach would be reusing the KNXnet/IP tunneling ACK to measure the time between sending the request to the gateway and getting the ACK. The measured time corresponds to the RTT plus some processing overhead at the KNXnet/IP gateway.

KNX scanning time Connection-oriented scanning of the KNX network is currently a time-consuming operation. As discussed previously, increasing the number of threads comes with the drawback of imprecise measurements. We feel that the implementation of connection-oriented measurement should be improved regarding the timeout value.

While applying Smart Eagle to the ABB smart grid demo lab, we usually observed that a connection-oriented device descriptor read takes less than 200 ms. Hence, if a device does not respond within 200 ms, the connection attempt should be aborted. This way, the single threaded device discovery time would be reduced to roughly one minute ($200\,ms \cdot 256\,devices$).

One way to do this is by adapting the Calimero library. To get even more control over the timing, one could record the UDP packets exchanged between control unit and gateway. Next, we determine which parts of the UDP packet need to be adapted (e.g. KNX destination address) and replay the protocol for different targets.

Contiki timer precision Due to the limited Contiki timer precision, our IEEE 802.15.4 RTT measurements have an error range of 80 microseconds. An increased timer precision would help to detect minor variations in RTT allowing to study the influence of different parameters, e.g. radio chip or network load.

We would replace the current driver with a version having microseconds accuracy. The hardware timer is configured in such a way that it gets incremented every microsecond ($currTimer$). A 64-bit unsigned integer counts the number of timer overflow interrupts ($\#overflows$). The time elapsed since the timer has been been initialized is obtained through the following calculation:

$$time \quad = \quad \#overflows \cdot maxTimer + currTimer$$

where $maxTimer$ defines the maximum timer value.

8.1.2 Extensions

Apart from improving the existing functionality, we have several ideas to extend our software towards more measurement functionality and increased network

[11] http://iperf.sourceforge.net/

support.

Traffic flow analysis In a heterogeneous network, traffic flows often across multiple different networks. Examples are the KNXnet/IP protocol or 6LoWPAN. For the Smart Eagle user, it would be beneficial to track network traffic across different networks to determine the communication path, find the source of packet loss or determine the bottleneck. We imagine the user clicking on a node in a network graph to highlight all communication partners. He shall even be able to select one packet and determine its path.

One way to achieve this is by installing measurement software on the gateways between the different networks. Yet, we propose a less intrusive mechanism based on sniffing. There are various ways to track packets, for example source and destination address, sequence number, payload or a combination.

Bandwidth measurement We have not yet implemented measurement functionality to get the bandwidth or utilization. A straight forward method would be deploying additional probes and send probing traffic between them. However, this opposes our goal of keeping a small footprint.

For IP networks, there are various other techniques to determine these parameters (Subsection 2.4.1). An example technique we belief can be adapted to our networks as well is based on RTT measurements and is called *variable packet size probing*. By varying the probing packet size, the RTT changes.

However, these techniques require a precise timer which we currently do not have for both networks types. Moreover, when adapting these techniques for our network types, a comprehensive study under various settings is required to confirm the results and possibly adapt the algorithms or mathematical models. The example of KNX illustrates the difficulty because we don't have a network layer RTT measurement. The process of establishing a connection in connection-oriented mode may very well render the variable packet size probing unusable because the RTT differences are statistically insignificant.

IP networks As measurements techniques for IP networks are well established, we suggest integrating these tools into Smart Eagle by developing an IP control unit. It would be a valuable extension as IP networks are common and used in smart grid environments as well. Furthermore, it would answer the question how well existing tools can be integrated into our architecture. We suggest starting with the *ping* utility as it is available on most systems.

Top down deployment We suggested various extensions to improve already implemented algorithms or to add new functionality. This kind of software evolution is normal, even in industrial products. As such measurement infrastructure is in place for many years, the question how to update such a system is important.

We propose a top down deployment approach: the measurement application fetches updates from the Internet as it is running on a PC. When the control

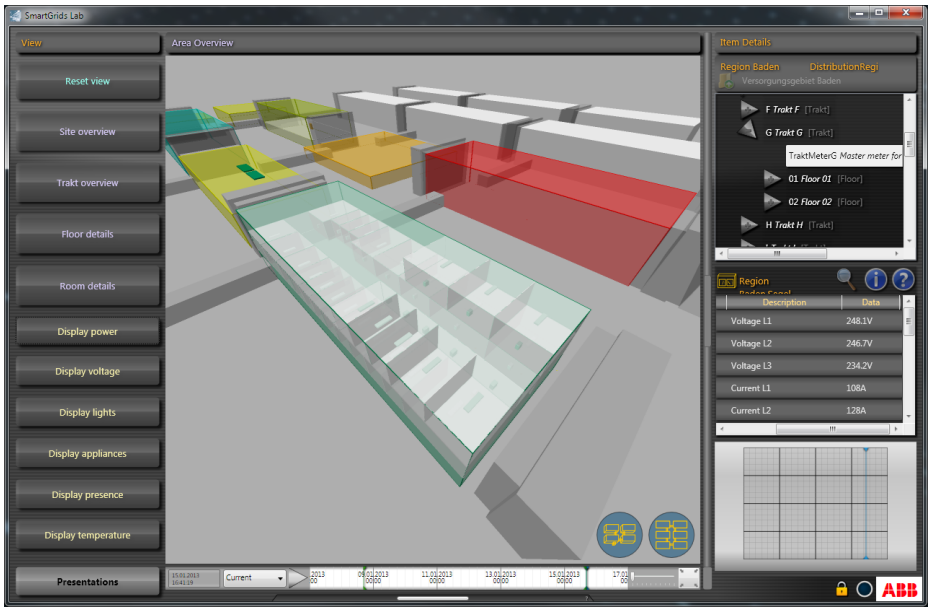

Figure 25: Screenshot of ABB smart grid demo lab 3D user interface. A discussion about integrating the Smart Eagle measurements into this GUI is ongoing.

units starts, it just launches a tiny loader application sending a beacon. When the beacon is received by the measurement application, it replies with its IP and the control unit can download the main program using HTTP. This way, the control unit runs always the most recent version without any user interaction.

GUI The visualize the long term network behavior, we suggest extending the GUI with various plots. A few use cases of plots against time:

- Traffic volume: we expect that the amount of traffic varies within one day. The plot helps to determine the time of peak load and maybe the traffic volume can be reduced during this time frame by certain optimizations.
- RTT behavior: we imagine this plot to be useful during development of network nodes. One could for example successively increase the CPU load on the network node and observe if and how this influences the RTT.
- Packet loss: determined by the number of missed RTTs while monitoring. For instance isolating the time period of high packet loss could help to determine the cause (e.g network load or interference from external devices).

Another GUI element which we consider helpful to optimize the daily work-flow is a network overview which highlights (potential) problems. This provides the user with a quick overview of elements requiring attention and avoids that he looses track.

8.1.3 Integration

Smart Eagle is currently a stand-alone, distributed system. In a realistic deployment, network measurements are combined with other relevant data for controlling the smart grid, for example power levels. During a research project at ABB corporate research Baden-Dättwil, a user interface with 3D building animation has been developed [52]. It is specially designed to display measurements (Figure 25) and a discussion about the integration of Smart Eagle is ongoing.

References

[1] F.R. Yu, Peng Zhang, Weidong Xiao, and P. Choudhury. Communication systems for grid integration of renewable energy resources. *Network, IEEE*, 25(5):22 –29, september-october 2011.

[2] H. Farhangi. The path of the smart grid. *Power and Energy Magazine, IEEE*, 8(1):18 –28, january-february 2010.

[3] Zhenhua Jiang, Fangxing Li, Wei Qiao, Hongbin Sun, Hui Wan, Jianhui Wang, Yan Xia, Zhao Xu, and Pei Zhang. A vision of smart transmission grids. In *Power Energy Society General Meeting, 2009. PES '09. IEEE*, pages 1 –10, july 2009.

[4] Q&a on the deployment of smart electricity grids and smart meters. `http://ec.europa.eu/energy/gas_electricity/smartgrids/doc/20110412_memo.pdf`. Retrieved on 03.08.2012.

[5] Massoud Amin and Phillip F. Schewe. Preventing blackouts: Building a smarter power grid. *Scientific American*, May 2007.

[6] Judy Lai Chris Marnay, Nicholas DeForest. A green prison: The santa rita jail campus microgrid. Technical report, Lawrence Berkeley National Laboratory, May 2012.

[7] Introduction to knx and konnex. `http://www.weinzierl.de/download/company/Knx_Info.pdf`.

[8] Knx. `www.knx.org`.

[9] W. Kastner, G. Neugschwandtner, S. Soucek, and H.M. Newmann. Communication systems for building automation and control. *Proceedings of the IEEE*, 93(6):1178 –1203, june 2005.

[10] Knx system specifications - volume 3: Architecture, June 2009. Version 3.

[11] Information technology - open systems interconnection - basic reference model: the basic model, November 1994.

[12] Knx system specifications - volume 6: Profiles.

[13] Eduardo TOVAR Anis KOUBAA, MÃ¡rio ALVES. Ieee 802.15.4 for wireless sensor networks: A technical overview. Technical report, Polytechnic Institute of Porto (ISEP-IPP), July 2005.

[14] Zigbee specification, zigbee document 053474r13, December 2006.

[15] Chen Yibo, Kun-Mean Hou, Haiying Zhou, Hong ling Shi, Xing Liu, Xunxing Diao, Hao Ding, Jian-Jin Li, and C. de Vaulx. 6lowpan stacks: A survey. In *Wireless Communications, Networking and Mobile Computing (WiCOM), 2011 7th International Conference on*, pages 1 –4, sept. 2011.

[16] Ieee standard for information technology– local and metropolitan area networks– specific requirements– part 15.4: Wireless medium access control (mac) and physical layer (phy) specifications for low rate wireless personal area networks (wpans). *IEEE Std 802.15.4-2006 (Revision of IEEE Std 802.15.4-2003)*, pages 1 –320, 7 2006.

[17] Ieee standard for information technology - telecommunications and information exchange between systems - local and metropolitan area networks - specific requirements. supplement to carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications - physical layer parameters and specifications for 1000 mb/s operation over 4-pair of category 5 balanced copper cabling, type 1000base-t. *IEEE Std 802.3ab-1999*, page i, 1999.

[18] R. S. Prasad, M. Murray, C. Dovrolis, K. Claffy, Ravi Prasad, and Constantinos Dovrolis Georgia. Bandwidth estimation: Metrics, measurement techniques, and tools. *IEEE Network*, 17:27–35, 2003.

[19] G. Kessler and S. Shepard. Rfc 1739: A primer on internet and tcp/ip tools. `http://tools.ietf.org/html/rfc1739`, December 1994.

[20] J. Postel. Rfc 792: internet control message protocol. `http://tools.ietf.org/html/rfc792`, September 1981.

[21] Stefan Savage. Sting: a tcp-based network measurement tool. In *Proceedings of the 2nd conference on USENIX Symposium on Internet Technologies and Systems - Volume 2*, USITS'99, pages 7–7, Berkeley, CA, USA, 1999. USENIX Association.

[22] Bernhard Hechenleitner Felix Strohmeier, Heinz Doerken. Aquila distributed qos measurement. In *In Proc. of COMOCON8 Conference*, pages 177–185, 2001.

[23] Richard Black, Austin Donnelly, and Cedric Fournet. Ethernet topology discovery without network assistance. In *Proceedings of the 12th IEEE International Conference on Network Protocols*, ICNP '04, pages 328–339, Washington, DC, USA, 2004. IEEE Computer Society.

[24] Kamal Ahmat. Ethernet topology discovery: A survey. *CoRR*, abs/0907.3095, 2009.

[25] Lorenzo Colitti. *Internet Topology Discovery Using Active Probing*. PhD thesis, UNIVERSITA DEGLI STUDI ROMA TRE, 2006.

[26] Utilityiq network element manager. `http://www.silverspringnet.com/pdfs/SilverSpring-Datasheet-UtilityIQ-NEM.pdf`.

[27] Intelligent metering network management. `http://www-01.ibm.com/software/tivoli/intelligent-metering/`.

[28] Georg Neugschwandtner and Wolfgang Kastner. Congestion control in building automation networks: Considerations for KNX. In *Proc. 35th Annual Conference of the IEEE Industrial Electronics Society (IECON 2009)*, pages 4149–4154, November 2009.

[29] S. Cavalieri. Analysing congestion in knxnet/ip communication system. In *Industrial Technology (ICIT), 2011 IEEE International Conference on*, pages 244 –249, march 2011.

[30] A. Vaccaro S. L. Ullo and G. Velotto. Performance analysis of ieee 802.15.4 based sensor networks for smart grids communications. *Journal of Electrical Engineering: Theory and Application*, 1-2010/Iss.3:129–134, 2010.

[31] Yu Yang, Peng Xia, Liang Huang, Quan Zhou, Yongjun Xu, and Xiaowei Li. Snamp: A multi-sniffer and multi-view visualization platform for wireless sensor networks. In *Industrial Electronics and Applications, 2006 1ST IEEE Conference on*, pages 1 –4, may 2006.

[32] Xin Kuang and Jianhua Shen. Snds: A distributed monitoring and protocol analysis system for wireless sensor network. In *Networks Security Wireless Communications and Trusted Computing (NSWCTC), 2010 Second International Conference on*, volume 2, pages 422 –425, april 2010.

[33] Killerbee: Framework and tools for exploiting zigbee and ieee 802.15.4 networks. `http://code.google.com/p/killerbee/`.

[34] M. Pipattanasomporn, H. Feroze, and S. Rahman. Multi-agent systems in a distributed smart grid: Design and implementation. In *Power Systems Conference and Exposition, 2009. PSCE '09. IEEE/PES*, pages 1 –8, march 2009.

[35] Diego Adolf. Smartscript a domain-specific language for appliance control in smart grids. Master's thesis, EEH - Power Systems Laboratory, Swiss Federal Institute of Technology (ETH) Zurich, 2011.

[36] Boris Malinowsky, Georg Neugschwandtner, and Wolfgang Kastner. Calimero: Next generation. In *Proc. KNX Scientific Conference 2007*, November 2007.

[37] Dominik Windhab. Bachelorthesis: Bluetooth - knx gateway. `http://www.androidpit.de/de/android/market/apps/app/Bluetoothsniffer.rc.de/KNX-RF-Sniffer`, 2008.

[38] Abb i-bus tool: A professional service tool for knx system integrators. `http://www.abb.com/cawp/seitp202/eec9ea9d970bf95dc125799f003bf7f4.aspx`, 2012.

[39] Daniel Lechner Harald Weillechner. Knxnet/ip wireshark dissector. `http://knxnetipdissect.sourceforge.net/index.html`.

[40] Abb i-bus tool: A professional service tool for knx system integrators. http://www.weinzierl.de.

[41] Product data sheet - sensor network analyzer. http://www.daintree.net/downloads/datasheets/daintree_sna.pdf.

[42] Smartrftm packet sniffer user manual swru187f. http://www.ti.com/tool/packet-sniffer, July 2011.

[43] Cisco Systems. Internet protocol (ip) multicast. Technical report, Cisco Systems, Inc, 2000.

[44] L. Ramadoss and J.Y. Hung. A study on universal serial bus latency in a real-time control system. In *Industrial Electronics, 2008. IECON 2008. 34th Annual Conference of IEEE*, pages 67 –72, nov. 2008.

[45] Maarten Damen. Plugwise unleashed - a document explaining the protocol used by plugwise products. http://www.maartendamen.com/2010/08/plugwise-unleashed-document-released/, August 2010.

[46] Mariano Alvira. Using the freescale mc1322x series arm7 processor with integrated 802.15.4. http://mc1322x.devl.org/, October 2012.

[47] Sourcery codebench lite edition. http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/.

[48] Mc1322x advanced zigbee - compliant soc platform for the 2.4 ghz ieee 802.15.4 standard reference manual, January 2012. Rev. 1.6.

[49] Atmel avr2025: Ieee 802.15.4 mac software package - user guide, May 2012. Rev. 8412D.

[50] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[51] Oracle forum: prevent jtree from collapsing - programming example. https://forums.oracle.com/forums/thread.jspa?messageID=6212248.

[52] M. Naef and E. Ferranti. Multi-touch 3d navigation for a building energy management system. In *3D User Interfaces (3DUI), 2011 IEEE Symposium on*, pages 113 –114, march 2011.

9 Appendix

The appendix is organized in three parts. In the first part, the setup and deployment is described for all components of Smart Eagle. Next, we document the protocol on the interconnect between the IEEE 802.15.4 network adapter and the IEEE 802.15.4 control unit. In the end, the protocols on the intermediate network (HTTP request and XML response) are described.

9.1 Setup and deployment

Setup and deployment is described with respect to the current directory structure of the Smart Eagle project. We start by describing the default gateway issues. Afterwards, we show how to setup Eclipse containing all projects as Smart Eagle is split into the following components: *library*, *IEEE 802.15.4 control unit*, *KNX control unit* and *measurement application*. Next, we describe for both control units and the measurement application the setup and launch procedure.

9.1.1 Default gateway

For the locator beacon service to work, a default gateway needs to be configured (Subsection 3.4.2). If no default gateway is present, the locator beacon service fails to start with an IOException (printed on console).

9.1.2 Eclipse setup

The Smart Eagle project folder contains four directories. The following folders contain an Eclipse project and need to be imported into one workspace:

- SmartEagleLibrary
- 802154Probe/802154ControlUnit
- knxProbe
- MeasurementApplication

The projects are cross referenced automatically in Eclipse.

9.1.3 IEEE 802.15.4 probe

The following steps are required to setup and launch the IEEE 802.15.4 control unit on Ubuntu Linux:

1. Compile the Smart Eagle application together with Contiki (Subsection 4.2)
2. Prepare automatic deployment by making the following scripts executable (path relative to 802.15.4 probe directory)
   (a) 802154ControlUnit/launch-contiki.sh

(b) contiki/examples/smart-eagle/deploy.sh

(c) contiki/examples/smart-eagle/deployInit_compile.sh

3. Compile the intermediate C program allowing to access the serial console with root permissions (Subsection 4.4.4) by executing the following script (it asks for the root password):
contiki/examples/smart-eagle/deployInit_compile.sh

4. Plug in the Econotag. Probably wait for a few minutes because sometimes the Ubuntu modem driver tries to connect to the Econotag.

5. Configure the correct ttyUSB device in the following "configuration file" (dedicated Java class containing static variables): src/knx/KNXConfig.java

6. Launch the control unit by executing the Java main class:
src/main/Main.java

9.1.4 KNX probe

There is no configuration required to launch the control unit. The main class is: src/main/Main.java

9.1.5 Measurement application

The main class to launch the measurement application is:
src/app/App.java

9.2 IEEE 802.15.4 interconnect protocol

This subsection describes the interconnect protocol between the IEEE 802.15.4 control unit and the Econotag network adapter running a Contiki based Smart Eagle process.

The data exchange is based on text and has the following format: <prefix>=<csv> where <prefix> is the name of a parameter or command and <csv> are comma separated values. Between the csv elements, there are no white spaces to reduce bandwidth requirements. To describe the <csv> part, we use the same basic building blocks (Table 11).

Input The input commands are summarized in Table 12. All commands received by the network adapter are handled in the following way:

1. Receive and parse command.

2. Execute command. The only possible result output is a *sniffack* as a response to a RTT measurement or a sweep. This is generated by the sniffer.

3. *Se*nd string "ack" *to c*onfirm that the command has been completed.

Output The sniffer sends information about all received frame to the control unit. The sniffed data can either be traffic between the devices or a response to a command. The protocol is described in Table 13.

9.3 Intermediate network protocols

The HTTP protocol for IEEE 802.15.4 and KNX are described in Table 14 and Table 15 respectively. In the tables, the request parameters and the expected output are summarized. As the XML schema can be generated automatically from the corresponding Java class containing the JAXB annotations, it is not part of this document.

Item	Description
<longSrc>	Set to “1” if the source address is a 64-bit address, set to “0” if it is a short 16-bit address.
<longDst>	Set to “1” if the destination address is a 64-bit address, set to “0” if it is a short 16-bit address.
<srcAddr>	Source address in 8-bit blocks, separated by white spaces
<dstAddr>	Destination address in 8-bit blocks, separated by white spaces
<srcPan>	Source pan ID (range: 0 - 65536)
<dstPan>	Destination pan ID (range: 0 - 65536)
<lqi>	Link quality (range: 0 - 255)
<seq>	Frame sequence number (range: 0 - 255)
<channel>	Channel number (range: 0 -15)
<subnet>	Subnet identifier (range: 0 -255)

Table 11: Elements to describe the interconnect protocol for the IEEE 802.15.4 probe.

Prefix	Argument	Description
setchannel	<channel>	Configure channel
setpanid	<srcPan>	Configure pan ID
setshortsrc	<srcAddr>	Configure short source address
setlongsrc	<srcAddr>	Configure long source address
ping	<dstAddr>	Perform RTT measurement
sweep	<subnet>	Perform sweep

Table 12: IEEE 802.15.4 probe interconnect protocol: commands from control unit to network adapter. Command execution is confirmed with a message containing “ack” as a String which is sent from the network adapter to the control unit.

Prefix	Content (csv)	Description
snif	<longSrc>, <srcAddr>, <longDst>, <dstAddr>, <srcPan>, <dstPan>, <seq>, <lqi>	Sniffed data frame
snifack	<seq>, <lqi>	Sniffed ack frame
pingack	<seq>, <lqi>	sniffed ack frame as a response to an RTT measurement

Table 13: IEEE 802.15.4 probe interconnect protocol: sniffer data output. Apart from a confirmation *ack* after successfully executing a command, the sniffer is the only entity generating output.

Request	Parameters	XML content
linkLayerRtt	destination=<dstAddr>	<srcAddr> response time (int) response dimension (String) response precision (String) <lqi> <seq> measurement timestamp (long) measurement ID (int)
sweep	destination=<dstAddr> Only the first 8 bits are processed because it denotes the subnet	Multiple linkLayerRtt responses
configure	One of the following parameters: setchannel=<channel> setpanid=<srcPan> setsourceaddress=<srcAddr>	No XML response, HTTP status code only
sniffer	none	Traffic type (Eight02154TrafficType) <srcAddr> <dstAddr> <srcPan> <dstPan> <seq> <lqi> measurement timestamp (long) measurement ID (int)
logs	none	Long entries (String)

Table 14: Protocol between measurement application and IEEE 802.15.4 control unit. All communication is through HTTP and the parameters are passed as part of the URL.

Request	Parameters	XML content
linkLayerRtt	destination=<iAddr>	positive ACK (boolean) success (boolean) response time (long) response dimension (String) measurement timestamp (long) transaction ID (int)
networLayerRtt	destination=<iAddr> connectionOriented=<boolean>	success (boolean) response time (long) response dimension (String) measurement timestamp (long) transaction ID (int)
sniffer	none	frame type (String) source address (String) destination address (String) frame length (short) frame length dimension (String) frame id (int) timestamp (long)
logs	none	Long entries (String)

Table 15: Protocol between measurement application and KNX control unit. All communication is through HTTP and the parameters are passed as part of the URL. An individual address is denoted with $<iAddr>$.